Master thesis

Multi-class Fork-Join queues & The stochastic knapsack problem

Sihan Ding

March, 2011

Supervisor UL:
Dr. Floske Spieksma

Supervisors CWI:
Drs. Chrétien Verhoef
Prof.dr. Rob van der Mei

Preface

This thesis is a result of my internship at the Centrum Wiskunde en Informatica (CWI). It is also an important part of my master study in Leiden University (UL).

First of all, I want to give my special thanks to Chrétien Verhoef for all the daily advices and help from the beginning to the very end. I came to CWI with very little pre-knowledge about queueing theory and programming. His patience and clear explanation have helped me a lot. Also, I want to thank Floske Spieksma, my supervisor in Leiden, who spent a lot of time to correct my thesis and provided me with her solid theoretic knowledge. I also want to thank Rob van der Mei, who gave me the opportunity to do my internship in CWI. Rob, Floske and Sandjai Bhulai always inspired me with their professional and insightful thinking. They are all very dedicated researchers. I learned not only from their knowledge, but also from their active working attitudes.

I want to thank all the colleagues in PNA2, and Bram de Vries who shared the office with me, all of you made this period of my life colorful and special. All those ping-pong playing times and interesting lunch talks are unforgettable for me. Through my experience in CWI, I re-define the term "research", because it brought so much joy and happiness everyday.

Last but not least, I want to thank my parents, who are loving me and supporting me all along. I also give my thanks to Stella, who always has faith in me and our future.

Sihan Ding

Amsterdam, March 2011

Summary

This thesis consists of two topics:

1. Performance analysis of multi-class Fork-Join queueing systems
2. Call admission control policy of stochastic knapsack problem

The multi-class Fork-Join queueing system is an extension of the single-class Fork-Join queueing system. In such a system, different types of jobs arrive, and then split into several sub-jobs. Those sub-jobs go to parallel processing queues. Sub-jobs from different types may go to the same queue, and we call this overlapping. There is a server in front of each queue. After all sub-jobs of one job are completed, they synchronize in the synchronization buffer and then leave the system. Many communication and networking systems can be modeled as such systems. In the literature, hardly any exact results are known for the expected sojourn time of the multi-class Fork-Join queueing systems, and neither for the synchronization time. In this thesis, we study the expected sojourn time for each job type. We further investigate the expected synchronization time of each job type in order to optimize the size of the synchronization buffer. We develop methods to approximate the expected sojourn time and the expected synchronization time. Through extensive numerical experiments, we show that our approximation method provides a close approximation of the sojourn time. Evaluation results also lead to highly accurate approximation of the optimal synchronization buffer size.

The stochastic knapsack model was first built as a model for cellular networks. Nowadays it is also used for modeling other resource-sharing communication networks. The objective is to derive an optimal call admission control policy. The theory of Markov decision processes can be applied to compute the optimal call admission control policy. However, a large state space causes computational complexity. Therefore, other policies are developed in the literature with less computational complexity, such as reservation and threshold policies. We compare these types of policies with regard to reward performance. We show through examples that the reservation policy can perform badly. We provide a method to improve the reservation policy in a medium size state space.

Contents

Part I

Multi-class Fork-Join queueing system

Chapter 1

Introduction to multi-class Fork-Join queues

Fork-Join queues arise quite often in computer networks and parallel processing systems. For example, a certain type of product has to undergo a number of operations requiring parallel processing in different machines. Once one operation is completed, the product will be put into the warehouse waiting for subsequent operations. As soon as all the operations of a product are completed, this product will be delivered and subsequently leave the warehouse. However, this Fork-Join queueing network has its limitations in practice. Consider the production example again. In reality, often there is more than one type of product. Different types of products may require processing on the same machine, and we call this overlapping. Analyzing a problem with multi-type products is more complicated than a problem with single-type products, but it has wide applications in industry. Therefore, we are motivated to extend the one type product problem to the multiple products problem, namely multi-class Fork-Join queues.

In this thesis, jobs are interpreted as products, and sub-jobs are interpreted as operations. In this model, each sub-job needs to wait in the buffer for a certain amount of time where synchronization can take place. As the number of job types increases, the synchronization time may increase as well. In the previous example, the waiting time in the warehouse might generate a certain penalty per time unit. Therefore, people are interested in minimizing this penalty by adjusting the service rate of the system. We will investigate this problem in this thesis.

1.1 Model description

In this section, we will describe the multi-class queueing network that we will study. N different classes of customers or jobs arrive according to a *Poisson process*. The arrival processes are independent with rates λ_i, i=1,2,...,N. Furthermore, there are M independent servers. The service times are exponentially distributed with rate μ_j, j=1,2,...,M. Each server has an infinite capacity queue. When a job of class i arrives, a pre-defined task matrix $T = (t_{ij})_{N \times M}$ prescribes

how it forks to different queues. The task matrix can be interpreted as a fixed policy that tells a class i customer what queues to go to. In particular, $t_{ij} = 1$ when a type i customer forks to queue j. The service discipline in each queue is *FIFO*. When a sub-job is finished, it leaves the server and waits in the synchronization buffer. When all sub-jobs of one given job are finished, they will synchronize and leave the buffer, the job is then considered to be completed.

The following matrix is an example of 2×3 task matrix of a Fork-Join queueing system with 2 classes and 3 queues. We show its corresponding system in Figure 1.1.

$$T = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

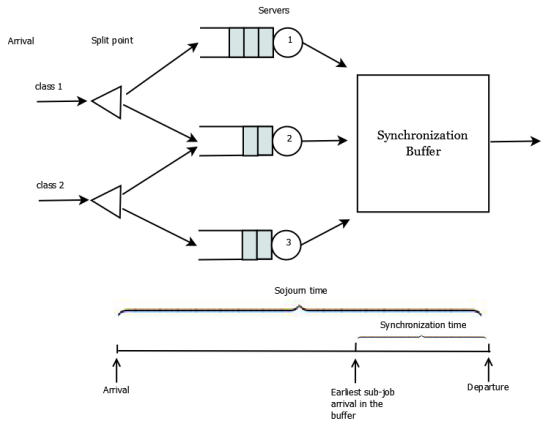


Figure 1.1: System corresponding to task matrix T

Before proceeding, we first give the formal definition of the sojourn time and synchronization time below.

Definition. *The sojourn time of a job is defined as the duration between its arrival and its departure from the synchronization buffer. The synchronization time of a job is defined as the time between the arrival of its first sub-job in the synchronization buffer and its last one.*

We are interested in the following question: *what is the average sojourn time of customers from class i?* For questions concerning the synchronization time, we will make the concepts more precise later on.

1.2 Related work

A lot of scientific effort has been put into understanding and analyzing Fork-Join queueing systems. This effort mainly focuses on the analysis of the sojourn time. When there is only one customer class, such a system is called a Fork-Join queueing system. An analytical expression for the sojourn time of the Fork-Join

queueing system is difficult to derive. In [10], Nelson and Tantawi derived such an expression for the homogeneous Fork-Join queueing model with two servers. Homogeneous means that all servers have the same service rate. However, in heterogeneous Fork-Join queues (i.e., where servers can have different service rates), the sojourn time is known to be intractable ([10]). However, there do exist approximation methods. In [9], different methods are used to approximate the average sojourn time of the system [5], and they extend their approximation to more general service time distributions. In [2] and [8], upper bounds and lower bounds for the sojourn time are derived.

Although there are various practical and theoretical results concerning one class Fork-Join queues, we have been unable to find any result for the extended system. As mentioned before, this type of system has many applications. Therefore, it is important to develop a method to approximate the sojourn time of multi-class queueing systems.

The analysis of the synchronization time provides the same difficulties as the sojourn time. We have not found any results concerning the synchronization time. However, we will show that it is an interesting topic with applications in the fields of buffer optimization.

1.3 Goals and structure

In this thesis, we will try to develop approximation methods for the sojourn time and the synchronization time in multi-class Fork-Join queueing systems. In an other words, we will give answer to the following questions:

1. What is the average sojourn time of the class i customer?
2. How long do class i customers on average wait in the buffer?

The thesis is structured as follows. Chapter 2 is divided into three sections. In section 2.1, we will first discuss an approximation method of the expected sojourn time, then we will evaluate the method numerically, and compare the results with computer experiments. In section 2.2, we will motivate why we are interested in the synchronization time. The method for approximating the synchronization time and its performance can be found in subsections 2.2.1 and in 2.2.2 respectively. The conclusion of the synchronization time analysis is in section 2.2.3. The future work is shown in section 2.3.

Chapter 2

Analysis of multi-class Fork-Join queues

In this chapter, we present an approximation method for the expected sojourn time as well as the expected synchronization time for the multi-class Fork-Join queue. After introducing the method, we evaluate its performance in different systems and under different scenarios. Performance is evaluated by comparing simulation results with approximations. The numerical results will show that our method provides a close upper bound of the real system.

2.1 Sojourn time analysis

In this thesis, we will use an approximation method based on the *Maximum Order Statistics (MOS)*. We will first build a new model based on the original model by assuming the independent arrival processes for the queues. Then we use *MOS* to derive an expression for the sojourn time of the new model. In [9], Lebrecht and Knottenbelt use *MOS* by assuming arrival independency to derive an upper bound for single-class Fork-Join queues. They conclude that *MOS* performs well in heterogeneous systems, and performs relatively badly in homogeneous systems. In this chapter, we will first give the definition of *MOS* and its formula by constructing a new model called the *MOS* model. Then, we will extend *MOS* method to the multi-class case. Then, we will compare its performance with simulation results in several systems. For each system, we consider both the homogeneous and heterogeneous cases. The numerical results for this approximation appear to be close to simulation result. Conclusions are given after the comparison.

2.1.1 The Maximum Order Statistic (MOS) model

For every multi-class Fork-Join queueing system, we can always construct its corresponding *MOS* model by taking the following steps:

First, the service rates of all servers remain the same. Second, take the arrival

process of each queue being independent with arrival rate $\bar{\lambda_i}, i = 1, 2, \ldots, M$. where:

$$(\bar{\lambda_1}, \bar{\lambda_2}, \ldots, \bar{\lambda_M}) = (\lambda_1, \lambda_2, \ldots, \lambda_N)' \cdot T.$$

This results in a new model with M independent *M/M/1* queues. The expected sojourn time of the *MOS* is used to approximate the expected sojourn time of original model. However, in this model, sub-jobs of one job may arrive at different time, while sub-jobs of one job arrive at the same time in original model. Therefore, there are time gaps between arrival of a job's sub-jobs. This leads to the following question: how to define the sojourn time in *MOS* model? We developed the following measurement of sojourn time in the *MOS* model. Each arriving job gets an *id* and a class type. The *id* is assigned in order of arrival, namely $id = 1$ for the first arrival, $id = 2$ for the second, etc. The class type is assigned according to the probability $P_i = \frac{\lambda_i}{\bar{\lambda_j}}$, where $\bar{\lambda_j}$ is the arrival rate to queue j, and λ_i is the arrival rate of class i jobs in the original system. After a job finishes service, it leaves the system. Figure 2.1 are an example of an multi-class Fork-Join model and its corresponding *MOS* model.

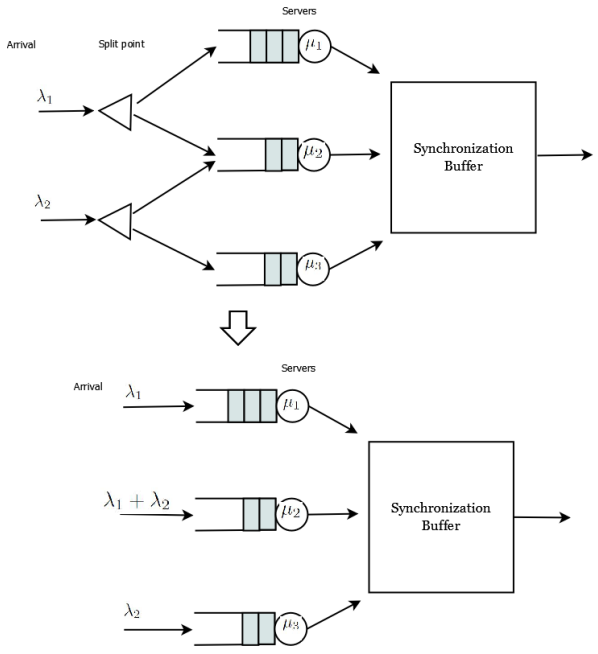


Figure 2.1: Multi-class Fork-Join model and corresponding *MOS* model

In the *MOS* model, sub-jobs of one job can arrive at different times, while in original model, sub-jobs of one job arrive at the same time. Therefore, in order to have a close approximation to the original sojourn time, it would be logical to ignore the time gap between arrival of sub-jobs that belong to one job. Therefore, we use the following way to measure the sojourn time of *MOS* model. After a job completes its service, we will record the response time (waiting time + service time) of this job. For those jobs with same *id* and same class type, we take the maximum response time as the sojourn time of this class type.

Since the arrival and service of each queue behaves independently, we can use the *Maximum Order Statistics* to derive an exact expression for the sojourn time of such system. We will first give the definition of *Maximum Order Statistics* (cf. [9]).

Definition. *Any finite sequence of random variables, X_1, X_2,$\cdots$ X_n can be ordered as $X_{(1)}$, $X_{(2)}$, $\cdots$ $X_{(n)}$, where $X_{(1)} \leq X_{(2)} \leq \cdots \leq X_{(n)}$. Then $X_{(1)}$, $X_{(2)}$, $\cdots$ $X_{(n)}$ are the order statistics of X_1, X_2,$\cdots$ X_n.*

We denote by X_i the sojourn time of ith queue in the stationary situation, and we let F_i represent the cumulative distribution function (cdf) of the sojourn time of ith queue in stationary situation, i.e. $X_i \sim \exp(\mu_i - \bar{\lambda_i})$, $F_i = \exp(\mu_i - \bar{\lambda_i})$, $i = 1, 2, \ldots, M$. The cdf of maximum order statistic $X_{(n)}$ can be calculated [9] by following formula:

$$F_{X_{(n)}}(x) = P(X_{(n)} \leq x) = P(X_{(1)} \leq x, X_{(2)} \leq x, \cdots, X_{(n)} \leq x).$$

By the previous assumption, we have:

$$F_{X_{(n)}}(x) = \prod_{i=1}^{n} F_i(x).$$

The mean sojourn time of $X_{(n)}$ is then

$$\mathbb{E}[X_{(n)}] = \int_{-\infty}^{+\infty} x \left(\sum_{i=1}^{n} \frac{f_i(x)}{F_i(x)} \right) \prod_{i=1}^{n} F_i(x)\, dx,$$

where $f_i(x)$ is the probability density function (pdf) of the sojourn time of ith queue in stationary distribution.

In [6], it is shown that:

$$\mathbb{E}[\max\{X_{(n)}\}] = \sum_{k=1}^{n} (-1)^{k+1} \sum_{(i_1,\ldots,i_k) \in S_k} \frac{1}{\mu_{i_1} + \cdots + \mu_{i_k}}, \tag{2.1}$$

where $S_k := \{(i_1, \ldots, i_k) : i_1, \ldots, i_k \in \{1, \ldots, N\}, i_1 < i_2 < \cdots < i_k\}$.

Formula (2.1) gives us an exact expression for the sojourn time of the *MOS* model. We apply this formula to approximate the original model.

2.1.2 Performance analysis

In this section, we will focus on analyzing the performance of *MOS* in multi-class Fork-Join queueing system for homogeneous and heterogeneous system respectively. In this thesis, the definition of homogeneous and heterogeneous is different from most of other papers in queueing theory. This is due to the fact that arrivals from the ith and jth class may both require processing by the kth server. This

means overlapping exists in some queues. For the purpose of comparison it would therefore be more convenient if we use following definitions.

Definition. *Homogeneous systems are systems for which all queues have the same work load, i.e.* $\rho_i = \rho$*, for all* $i = 1, 2, \ldots, M$*. Heterogeneous systems are systems for which at least one queue have a different work load, i.e. there exist* $i, j, i \neq j$*, such that* $\rho_i \neq \rho_j$*, where* $\rho_i = \frac{\lambda_i}{\mu_i}$*,* $i = 1, 2, \ldots, M$*.*

Before proceeding, we will list the notation that will be used in this section in Table 2.1.

Variable	*Description*
ρ_j	Work load of the jth queue
λ_i	Arrival rate of the ith customer
μ_j	Service rate of the jth server
$\mathbb{E}[S_i^{(sim)}]$	Expected sojourn time of ith class customer acquired by simulation
$\mathbb{E}[S_i^{(app)}]$	Expected sojourn time of ith class customer by using the *MOS* approximation
ϵ_i	Relative difference (error) between simulation and approximation of ith class

Table 2.1: Notation of sojourn time analysis

According to the definition of ϵ_i, we have:

$$\epsilon_i = 100\% \times |\frac{\mathbb{E}[S_i^{(app)}] - \mathbb{E}[S_i^{(sim)}]}{\mathbb{E}[S_i^{(sim)}]}|.$$

Here below, we will first show two systems as examples, and evaluate the performance of *MOS* method under such systems. System 1 is a simple system with 2 job classes and 3 servers. In order to gain more insight of the performance, system 2 is a more complicated system with 3 job classes and 8 servers. Then we show more systems in a structured way to study the influence of number of servers on performance.

System 1

We first analyze Fork-Join queueing system with 2 classes and 3 servers with $\lambda_1 = \lambda_2 = 1$. (see Figure 2.2)

For this system, we will investigate the performance of both the homogeneous case (Table 2.2) and the heterogeneous case (Table 2.3). In the homogeneous case, the load ρ varies between 0.1 and 0.9. In the heterogeneous case, we let ρ_1, ρ_2, ρ_3 take the values 0.25, 0.5 and 0.75, standing for *"Low", "Medium",* and *"High"* work load respectively.

System 1: homogeneous case

Since the work load for each queue is the same, this queueing system is symmetric. This leads to the same sojourn times for class 1 and class 2. As we can see

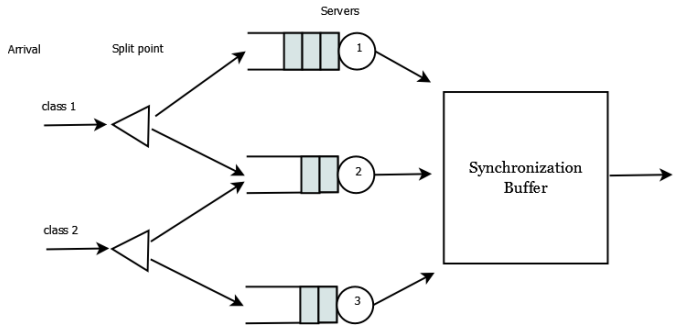


Figure 2.2: Performance analysis system 1

ρ	(λ_1, λ_2)	$\mathbb{E}[S_1^{(sim)}]$	$\mathbb{E}[S_1^{(app)}]$	$\mathbb{E}[S_2^{(sim)}]$	$\mathbb{E}[S_2^{(app)}]$	$\varepsilon_1\%$	$\varepsilon_2\%$
0.1	(1, 1)	0.129	0.130	0.128	0.130	0.78%	1.56%
0.2	(1, 1)	0.289	0.292	0.288	0.292	1.04%	1.39%
0.3	(1, 1)	0.494	0.505	0.496	0.505	2.23%	1.81%
0.4	(1, 1)	0.767	0.778	0.767	0.778	1.43%	1.43%
0.5	(1, 1)	1.140	1.167	1.145	1.167	2.37%	1.92%
0.6	(1, 1)	1.706	1.744	1.710	1.744	2.23%	1.99%
0.7	(1, 1)	2.627	2.713	2.653	2.713	3.27%	2.26%
0.8	(1, 1)	4.481	4.667	4.490	4.667	4.15%	3.94%
0.9	(1, 1)	10.082	10.501	10.041	10.501	4.16%	4.58%

Table 2.2: Performance analysis of homogeneous Fork-Join queue with 2 classes and 3 servers

(Table 2.2) the *MOS* approximation is quite close to the original system, with largest error less than 5% for all cases. Another observation is that *MOS* performs better with lower work load than with higher work load.

System 1: heterogeneous case

(ρ_1, ρ_2, ρ_3)	(λ_1, λ_2)	$\mathbb{E}S_1^{(sim)}$	$\mathbb{E}S_1^{(app)}$	$\mathbb{E}S_2^{(sim)}$	$\mathbb{E}S_2^{(app)}$	$\varepsilon_1\%$	$\varepsilon_2\%$
(0.25, 0.5, 0.75)	(1, 1)	0.622	0.633	3.047	3.074	1.77%	0.89%
(0.75, 0.25, 0.5)	(1, 1)	2.990	3.009	1.017	1.024	0.64%	0.69%
(0.5, 0.75, 0.25)	(1, 1)	1.855	1.899	1.550	1.560	2.37%	0.65%

Table 2.3: Performance analysis of the heterogeneous Fork-Join queue with 2 classes and 3 servers

Table 2.3 shows that the *MOS* performs well with maximum error 2.37% for the expected sojourn time of all classes. Comparing Tables 2.2 and 2.3, we can see that *MOS* performs better in the heterogeneous case than in the homogeneous with in general a smaller error.

System 2

In the previous system, we fixed $\lambda = 1$. However, what we see quite often in reality

is that different classes have different arrival rates. Therefore, we are interested in how different arrival rates influence performance of the approximation. In this system (Figure 2.3) we will analyze a system that has 3 customer classes with different arrival rates ($\lambda_1 = 1, \lambda_2 = 2, \lambda_3 = 4$). Furthermore, in order to investigate the performance of the *MOS* method in more complicated systems, we add more queues for each class, and more overlapping of queues between different classes.

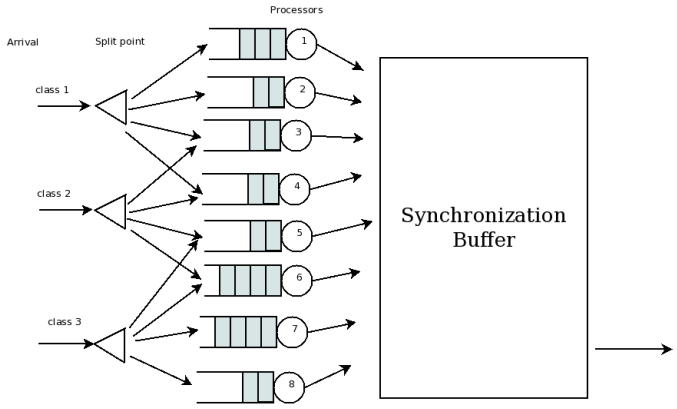


Figure 2.3: Performance analysis system 2

In this system, we investigate the performance of three cases. The homogeneous case (Table 2.4), the structured heterogeneous case (Table 2.5) and *General* case with 10 sets of random numbers are considered. In the homogeneous case, the work load varies from 0.1 to 0.9. In the structured heterogeneous case, the ρ_i are linearly located in the open interval (0,1), i.e. $\rho_1 = \frac{1}{9}, \rho_2 = \frac{2}{9}, \ldots, \rho_8 = \frac{8}{9}$ in the first scenario, and $\rho_1 = \frac{8}{9}, \ldots, \rho_8 = \frac{1}{9}$ in the second scenario. In the *General* case, we used Matlab to generate 10 sets of random numbers from the uniform distribution $\mathbb{U}(0, 1)$. The 10 sets of random numbers stand for 10 sets of different work loads. The random numbers are given in Table 2.6. The corresponding values of the per-class expected sojourn times are shown in Table 2.7.

System 2: homogeneous case

ρ	λ	$\mathbb{E}[S_1^{(sim)}]$	$\mathbb{E}[S_1^{(app)}]$	$\mathbb{E}[S_2^{(sim)}]$	$\mathbb{E}[S_2^{(app)}]$	$\mathbb{E}[S_3^{(sim)}]$	$\mathbb{E}[S_3^{(app)}]$	$\varepsilon_1\%$	$\varepsilon_2\%$	$\varepsilon_3\%$
0.1	(1, 2, 4)	0.172	0.173	0.061	0.061	0.048	0.049	0.58%	0.00%	2.08%
0.2	(1, 2, 4)	0.382	0.390	0.135	0.138	0.108	0.111	2.09%	2.22%	2.78%
0.3	(1, 2, 4)	0.653	0.669	0.229	0.236	0.182	0.190	2.45%	3.06%	4.40%
0.4	(1, 2, 4)	1.007	1.040	0.353	0.367	0.280	0.295	3.28%	3.97%	5.36%
0.5	(1, 2, 4)	1.486	1.561	0.524	0.550	0.415	0.443	5.05%	4.96%	6.75%
0.6	(1, 2, 4)	2.202	2.341	0.781	0.825	0.613	0.664	6.31%	5.63%	8.32%
0.7	(1, 2, 4)	3.455	3.641	1.199	1.283	0.940	1.033	5.38%	7.01%	9.89%
0.8	(1, 2, 4)	5.735	6.243	2.028	2.200	1.607	1.771	8.86%	8.48%	10.21%
0.9	(1, 2, 4)	12.797	14.061	4.486	4.955	3.601	3.987	9.88%	10.45%	10.72%

Table 2.4: Performance analysis of homogeneous Fork-Join queue with 3 classes and 8 servers

From Table 2.4, we see that in this more complicated system, the error in general is larger than in system 1. However, the performance is still acceptable with largest error being 10.72%. In the scenario of a low work load, *MOS* has a error

less than 5%.

System 2: heterogeneous case

ρ	λ	$\mathbb{E}[S_1^{(sim)}]$	$\mathbb{E}[S_1^{(app)}]$	$\mathbb{E}[S_2^{(sim)}]$	$\mathbb{E}[S_2^{(app)}]$	$\mathbb{E}[S_3^{(sim)}]$	$\mathbb{E}[S_3^{(app)}]$	$\varepsilon_1\%$	$\varepsilon_2\%$	$\varepsilon_3\%$
1/9∼ 8/9	(1, 2, 4)	0.450	0.462	0.502	0.525	2.214	2.293	2.67%	4.58%	3.57%
8/9∼ 1/9	(1, 2, 4)	8.784	9.083	0.799	0.838	0.177	0.183	3.40%	4.88%	3.39%

Table 2.5: Performance analysis of heterogeneous Fork-Join queue with 3 classes and 8 servers

Comparing Table 2.4 with Table 2.5, we can observe that *MOS* performs better in the heterogeneous case than in the homogeneous case. This is in accordance with system 1.

General cases

Set	ρ
1	(0.158, 0.971, 0.957, 0.485, 0.800, 0.142, 0.422, 0.916)
2	(0.792, 0.960, 0.656, 0.036, 0.849, 0.934, 0.679, 0.758)
3	(0.743, 0.392, 0.656, 0.171, 0.706, 0.032, 0.277, 0.046)
4	(0.097, 0.824, 0.695, 0.317, 0.950, 0.034, 0.439, 0.382)
5	(0.766, 0.795, 0.187, 0.490, 0.446, 0.646, 0.709, 0.755)
6	(0.276, 0.680, 0.655, 0.163, 0.119, 0.498, 0.960, 0.340)
7	(0.585, 0.224, 0.751, 0.255, 0.506, 0.699, 0.891, 0.959)
8	(0.547, 0.139, 0.149, 0.258, 0.841, 0.254, 0.814, 0.244)
9	(0.929, 0.350, 0.197, 0.251, 0.616, 0.473, 0.352, 0.831)
10	(0.585, 0.550, 0.917, 0.286, 0.757, 0.754, 0.380, 0.568)

Table 2.6: ρ for 8 queues drawing from $\mathbf{U}(0,1)$

Set	λ	$\mathbb{E}[S_1^{(sim)}]$	$\mathbb{E}[S_1^{(app)}]$	$\mathbb{E}[S_1^{(sim)}]$	$\mathbb{E}[S_2^{(app)}]$	$\mathbb{E}[S_1^{(sim)}]$	$\mathbb{E}[S_3^{(app)}]$	$\varepsilon_1\%$	$\varepsilon_2\%$	$\varepsilon_3\%$
1	(1, 2, 4)	31.411	34.699	7.281	7.524	2.838	2.854	10.47%	3.34%	0.56%
2	(1, 2, 4)	23.345	24.314	2.600	2.708	2.636	2.793	4.15%	4.15%	5.96%
3	(1, 2, 4)	3.062	3.090	0.775	0.791	0.414	0.419	0.91%	2.06%	1.21%
4	(1, 2, 4)	4.551	4.781	3.263	3.333	3.146	3.201	5.05%	2.15%	1.75%
5	(1, 2, 4)	5.046	5.375	0.477	0.490	1.025	1.087	6.52%	2.73%	6.05%
6	(1, 2, 4)	2.287	2.303	0.663	0.671	5.836	5.959	0.70%	1.21%	2.11%
7	(1, 2, 4)	1.792	1.849	1.103	1.130	6.132	6.415	3.18%	2.45%	4.62%
8	(1, 2, 4)	1.232	1.235	0.892	0.896	1.401	1.489	0.24%	0.45%	6.28%
9	(1, 2, 4)	12.951	13.180	0.338	0.349	1.224	1.290	1.77%	3.25%	5.39%
10	(1, 2, 4)	4.192	4.285	3.795	3.800	0.792	0.852	2.22%	0.13%	7.58%

Table 2.7: Performance analysis of homogeneous Fork-Join queue with 3 classes 8 servers

Since the work loads are all random numbers, no regularity is observed in this case (Table 2.7). However, we can see that *MOS* has acceptable performance with a maximum error being 10.47%. We notice that this maximum error occurs in type 1 jobs of set 1. In Table 2.6, we can see that service rates of two servers of type 1 jobs in set 1 are 0.971 and 0.957. These high work loads result in a high error.

More systems

In order to gain more insight of influence of the number of servers on sojourn time. We consider the 3 more systems with more servers and more overlapping. The systems are constructed in the following way. We consider 2 types of jobs with $\lambda_1 = 1, \lambda_2 = 2$ in all 3 systems. The first system is a queueing system of 2 classes and 9 servers with 3 servers overlapping. The second system is a queueing system of 2 classes 12 servers with 4 servers overlapping. The final system is a queueing system of 2 classes and 15 servers with 5 server overlapping. As we did before, for each system, we will evaluate the *MOS* approximation in the homogeneous case and the heterogeneous case with different work load scenarios.

ρ	λ	$\mathbb{E}[S_1^{(sim)}]$	$\mathbb{E}[S_1^{(app)}]$	$\mathbb{E}[S_2^{(sim)}]$	$\mathbb{E}[S_2^{(app)}]$	$\varepsilon_1\%$	$\varepsilon_2\%$
0.1	(1, 2)	0.206	0.209	0.114	0.116	1.46%	1.75%
0.2	(1, 2)	0.458	0.470	0.253	0.262	2.62%	3.56%
0.3	(1, 2)	0.772	0.806	0.428	0.448	4.40%	4.67%
0.4	(1, 2)	1.190	1.252	1.653	0.698	5.20%	6.89%
0.5	(1, 2)	1.758	1.878	0.964	1.047	5.21%	8.60%
0.6	(1, 2)	2.590	2.804	1.428	1.573	8.26%	10.15%
0.7	(1, 2)	3.972	4.367	2.186	2.454	9.94%	12.26%
0.8	(1, 2)	6.705	7.512	3.659	4.186	12.40%	14.40%
0.9	(1, 2)	15.070	17.072	8.308	9.514	13.28%	14.52%

Table 2.8: Performance analysis of homogeneous Fork-Join queue with 2 classes and 9 servers

ρ	λ	$\mathbb{E}[S_1^{(sim)}]$	$\mathbb{E}[S_1^{(app)}]$	$\mathbb{E}[S_2^{(sim)}]$	$\mathbb{E}[S_2^{(app)}]$	$\varepsilon_1\%$	$\varepsilon_2\%$
1/10∼ 9/10	(1, 2)	0.798	0.835	5.020	5.269	4.64%	4.96%
9/10∼ 1/10	(1, 2)	9.649	10.507	0.668	0.709	8.89%	6.14%

Table 2.9: Performance Analysis of heterogeneous Fork-Join queue with 2 classes and 9 servers

ρ	λ	$\mathbb{E}[S_1^{(sim)}]$	$\mathbb{E}[S_1^{(app)}]$	$\mathbb{E}[S_2^{(sim)}]$	$\mathbb{E}[S_2^{(app)}]$	$\varepsilon_1\%$	$\varepsilon_2\%$
0.1	(1, 2)	0.232	0.235	0.127	0.130	1.29%	2.36%
0.2	(1, 2)	0.511	0.529	0.280	0.291	3.52%	3.93%
0.3	(1, 2)	0.863	0.909	0.470	0.499	5.33%	6.17%
0.4	(1, 2)	1.323	1.412	0.717	0.777	6.73%	8.37%
0.5	(1, 2)	1.961	2.117	1.060	1.165	7.96%	9.91%
0.6	(1, 2)	2.851	3.161	1.569	1.751	10.87%	11.60%
0.7	(1, 2)	4.391	4.924	2.411	2.733	12.14%	13.36%
0.8	(1, 2)	7.464	8.469	4.037	4.661	13.46%	15.46%
0.9	(1, 2)	16.755	19.248	8.921	10.592	14.88%	18.73%

Table 2.10: Performance analysis of homogeneous Fork-Join queue with 2 classes and 12 servers

Based on the evaluation of these 3 systems, we can see that the performance of the *MOS* method is relatively bad while the queueing system has more servers. Specifically, in the system with 15 servers, the largest error is appeared in Table 2.12. However, in all the heterogenous cases, the *MOS* method still performs

ρ	λ	$\mathbb{E}[S_1^{(sim)}]$	$\mathbb{E}[S_1^{(app)}]$	$\mathbb{E}[S_2^{(sim)}]$	$\mathbb{E}[S_2^{(app)}]$	$\varepsilon_1\%$	$\varepsilon_2\%$
1/13∼ 12/13	(1, 2)	0.889	0.938	6.882	7.141	5.51%	3.76%
12/13∼ 1/13	(1, 2)	13.617	14.270	0.760	0.810	7.70%	6.58%

Table 2.11: Performance analysis of heterogeneous Fork-Join queue with 2 classes and 12 servers

ρ	λ	$\mathbb{E}[S_1^{(sim)}]$	$\mathbb{E}[S_1^{(app)}]$	$\mathbb{E}[S_2^{(sim)}]$	$\mathbb{E}[S_2^{(app)}]$	$\varepsilon_1\%$	$\varepsilon_2\%$
0.1	(1, 2)	0.252	0.257	0.137	0.140	1.98%	2.19%
0.2	(1, 2)	0.557	0.578	0.302	0.315	3.77%	4.30%
0.3	(1, 2)	0.940	0.991	0.506	0.539	5.43%	6.52%
0.4	(1, 2)	1.432	1.540	0.771	0.840	7.54%	8.95%
0.5	(1, 2)	2.112	2.310	1.132	1.259	9.37%	11.22%
0.6	(1, 2)	3.086	3.484	1.668	1.893	12.90%	13.49%
0.7	(1, 2)	4.715	5.372	2.557	2.954	13.93%	15.53%
0.8	(1, 2)	8.012	9.240	4.333	5.037	15.33%	16.25%
0.9	(1, 2)	17.808	21.000	9.824	11.448	17.92%	16.53%

Table 2.12: Performance analysis of homogeneous Fork-Join queue with 2 classes and 15 servers

ρ	λ	$\mathbb{E}[S_1^{(sim)}]$	$\mathbb{E}[S_1^{(app)}]$	$\mathbb{E}[S_2^{(sim)}]$	$\mathbb{E}[S_2^{(app)}]$	$\varepsilon_1\%$	$\varepsilon_2\%$
1/16∼ 15/16	(1, 2)	0.956	1.022	8.662	9.018	6.90%	4.11%
15/16∼ 1/16	(1, 2)	17.283	18.031	0.824	0.893	4.33%	8.37%

Table 2.13: Performance analysis of heterogeneous Fork-Join queue with 2 classes and 15 servers

well with largest error less than 9%. Furthermore, in homogeneous cases with low work load ($\rho \leq 0.5$), the performance of *MOS* method is acceptable with largest error being 11.22%.

2.1.3 Conclusion

According to the performance results, in general, the *MOS* method provides a tight upper bound of the real multi-class Fork-Join queueing system. However, in the homogeneous systems, this method performs better in low work load than in high work load. In addition, the *MOS* method performs well in a multi-class Fork-Join queueing system with up to 4 servers for each job class with maximum relative difference less than around 10%. We extended to system with more servers. The numerical results show that the *MOS* performs relatively badly in systems with up to 10 servers for each job class with a maximum relative difference less than around 18%. This confirms also experiments for single-class Fork-Join queues ([9]). Therefore, we conclude that this method performs better in queueing systems with less servers. In the heterogeneous systems, the *MOS* method has a close approximation to the real system regardless the number of servers. Furthermore, through all the systems with different arrival rates, it seems that the arrival rates of the jobs have very little influence to the performance of the *MOS* method.

We also conjecture that *MOS* will provide a good approximation if we allow general service times. This conjecture is based on [9]: we have experimentally found that the *MOS* methods extends to more general distributions, e.g. Erlang distribution, and will have better performance than exponential distribution.

Besides the tightness and the potential to extend to general distributions, an other advantage of *MOS* is that it is easy to implement.

2.2 Synchronization time analysis

In this section we will study the synchronization time for the multi-class Fork-Join queueing network. In the literature we have not found any existing result on the synchronization time in Fork-Join queueing systems. Therefore we will first explain the purpose of considering the synchronization time. Then we will develop a method to approximate the synchronization time. At last we will evaluate this approximation.

So far, people have been mainly interested in the sojourn time of the Fork-Join queueing system. Measuring of the occupancy of the synchronization buffer is also an important thing to consider. Take data transmission as an example. Data is transmitted through different networks (servers). When sub-data from one network is completed, it will wait in the synchronization buffer for other sub-data to complete. Data are from different hosts. As the number of hosts increases, the buffer size also increases. This means that in order to handle many different types of data, the buffer needs to be very large. This large buffer will require high cost for construction and maintenance. This motivates us to think about a flexible way of constructing the system, such that the buffer does not have to be infinite

large and data transmission is still at a satisfactory speed. Another example is in computer networks, where sub-jobs are processed by different processors. Then those sub-jobs wait in the memory (synchronization buffer). However, the memory has limited size, i.e. when too many sub-jobs wait in the memory, it will be fully occupied. It is important to consider a method to construct the system that can reduce the occupancy of the memory. We will illustrate this in more detail by the following example.

Consider the Fork-Join example in Figure 2.4. The first question arising to us is the relation between service rate and synchronization time. To study this, we do the following experiment: fix $\lambda = 1, \mu_1 = 2$, and we simulate this system for different values of μ_2. The synchronization time of type one jobs and its 95% confidence interval are plotted in Figure 2.5. As a matter of fact, μ_3 has no influence to the synchronization time of type one jobs. Therefore, the expected synchronization time of type one jobs only depend on μ_2.

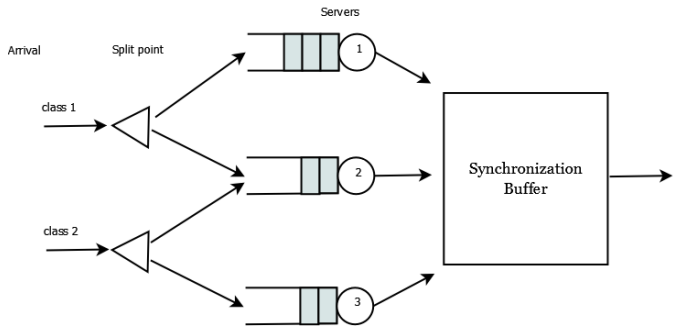


Figure 2.4: A simple 2 classes and 3 servers Fork-Join queueing system

The graph in Figure 2.5 shows that the synchronization time first decreases and then increases when μ_2 increases from 1.5 to 9. The graph has a unique minimum μ_2^*.

As a surprising conclusion, it is not always beneficial to take the service rates of other servers as large as possible, while one server has a fixed service rate. It is also interesting to observe from figure 2.5 that the minimum synchronization time is not achieved by taking $\mu_2 = \mu_1$.

We provide an intuitive argument. When μ_2 is relatively small, jobs in the first queue execute much faster than jobs in the second queue. Therefore, sub-jobs in the first queue have to wait in the buffer for the sub-jobs in the second queue. This leads to a decrease of the synchronization time. As μ_2 starts to increase, sub-jobs from queue 1 wait shorter in the buffer. However, as μ_2 continues to increase, the situation reverses: queue 2 jobs execute faster, and they have to wait for the first sub-jobs in the buffer. The faster the second server becomes, the longer sub-jobs from the second queue have to wait. This leads to an increase of the sojourn time. So, followed by this intuition, there exists an optimal point μ_2^*, such that:

$$\min_{\mu_2} \mathbb{E}[T_{syn}(\mu_2)] = \mathbb{E}[T_{syn}(\mu_2^*)],$$

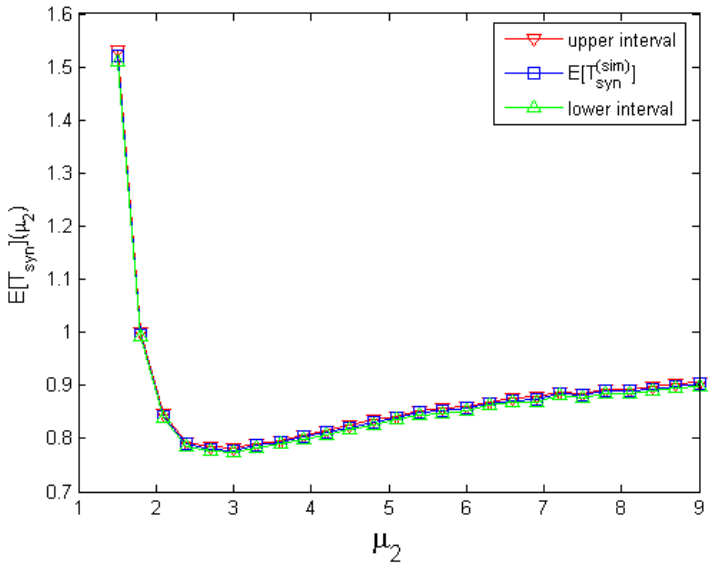


Figure 2.5: Expected synchronization time of type one jobs as a function of μ_2 for the model in Figure 2.4 ($\mu_1 = 2, \lambda_1 = 1$)

According to the definition of the synchronization time (T_{syn}), we have

$$T_{syn} = \max\{S_1, S_2\} - \min\{S_1, S_2\},$$

In general, for a system with n servers

$$T_{syn} = \max_{1 \le i \le n} \{S_i\} - \min_{1 \le i \le n} \{S_i\}. \tag{2.2}$$

Now the objective is to find the minimum synchronization time and its corresponding service rates. However, the difficulty to solve the above equation is caused by dependence of the queues. Therefore, similarly to the sojourn time analysis, we want to find an approximation for the minimum. In the next subsection, we will use the *order statistic* to approximate T_{syn}.

2.2.1 Order statistic (OS) approximation

In section 3.1, we defined the *Maximum Order Statistic*. Now we will use it together with *Minimum Order Statistics* to approximate the synchronization time (T_{syn}).

If we assume independent arrival processes and exponential service times distributions, the queues are independent M/M/1 queues, for which it is known that the sojourn time S has an exponential distribution with mean $\mathbb{E}[S_i] = \frac{1}{\mu_i - \lambda_i}, i = 1, 2, \ldots, M$.

The memoryless property of the exponential distribution has the following consequence.

Property. *In a system with M independent queues,*

$$\mathbb{E}[\min\{S_1, S_2, \ldots, S_M\}] = \frac{1}{(\mu_1 - \lambda_1) + (\mu_2 - \lambda_2) + \cdots + (\mu_M - \lambda_M)}. \quad (2.3)$$

Therefore, in formula (2.2), the part maximum sojourn time $\max_{1\leq i\leq n}\{S_i\}$ is approximated by formula (2.1), and the part minimum sojourn time $\min_{1\leq i\leq n}\{S_i\}$ is approximated by formula (2.3). We derive the following approximation for T_{syn}.

$$\begin{aligned} T'_{syn} =& (\sum_{k=1}^{M}(-1)^{k+1} \sum_{(i_1,\ldots,i_k)\in S_k} \frac{1}{\mu_{i_1} - \lambda_{i_1} + \cdots + \mu_{i_k} - \lambda_{i_k}}) - \\ & \frac{1}{(\mu_1 - \lambda_1) + (\mu_2 - \lambda_2) + \cdots + (\mu_M - \lambda_M)}. \end{aligned} \quad (2.4)$$

where S_k is defined in formula (2.1).

This yields the following unconstrained continuous optimization problem:

$$\min_{\mu_2,\cdots,\mu_M} T'_{syn}(\mu_2, \mu_3, \cdots \mu_M).$$

Thus, the approximation of μ^* is obtained by minimizing this continuous function.

Remark. *Similar to the sojourn time analysis, formula (2.4) does not calculate the synchronization time of the independent system. This is because of the time gap between arrival of sub-jobs. Formula (2.4) does not take this time gap into consideration.*

Using formula (2.4) to system in Figure 2.4, we plot our *OS* approximation curve and the simulation results and 95% confidence interval in graph 2.6.

2.2.2 Performance analysis

In this subsection, we evaluate the quality of the *OS* approximation for several systems. By comparing the simulation result and *OS* approximation, we will show that the *OS* is a good method for approximating the real optimum. We will also compute differences in the sojourn time when we compare large $\mu_i, i = 2, 3, \ldots, M$ and the optimum one.

We will evaluate the performance of the *OS* method in five systems. In the first three systems, we fix the service rate of one server, and vary the service rates of the other servers. In the last two systems, two of the service rates are fixed. Furthermore, we will show through these experimental results that the optimum expected synchronization time in *OS* approximation is achieved by putting the

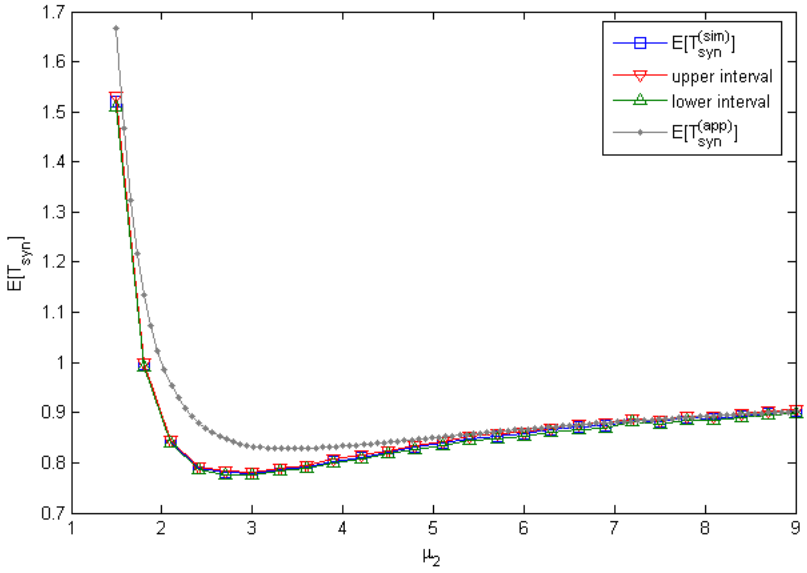


Figure 2.6: Expected synchronization time simulation comparing *OS* approximation

unfixed service rates to be equal. We will give the proof for this in system 2 and system 5. The proof for system 5 will also lead to the proof for system 3.

All the systems are constructed in the following way: we consider 9 scenarios by varying the work load of the first queue ρ_1 from 0.1 to 0.9. The arrival rates at all servers are the same, namely $\lambda = 1$. For each scenario, we show the simulation and the approximation results. Besides these two numerical results, we will also compare the effect on the sojourn time and the synchronization time when take the limit of the non-fixed service rates to infinity. In the performance of synchronization time in this thesis, we only consider one type of jobs. The synchronization time of other types of jobs is approximated through the same method.

Before we proceed, the notation of this section is shown in Table 2.14.

According to the notation, we have

$$\Delta\% = 100\% \times \frac{\mathbb{E}[T_{syn}(\mu^*)] - \mathbb{E}[T_{syn}(\mu^{*(os)})]}{\mathbb{E}[T_{syn}(\mu^*)]}$$

System 1 ($N = 1, M = 2$, for fixed μ_1)

We first consider the simplest system (Figure 2.7).

Apply formula (2.4) to this system, then we get:

Variable	*Description*
λ	Arrival rate
μ_i	Service rate of server i
μ^*	Optimal service rate acquired from simulation
$\mu_i^{*(os)}$	Optimal service rate of server i in OS approximation
$\mathbb{E}[T_{syn}(\mu)]$	Expected synchronization time by set μ as service rate
$\mathbb{E}[S(\mu)]$	Expected sojourn time by set μ as service rate
$\Delta\%$	Relative difference of the expected synchronization time between real optimum and the OS approximation
γ_i	$\mu_i - \lambda$

Table 2.14: Notation for performance analysis for the synchronization time

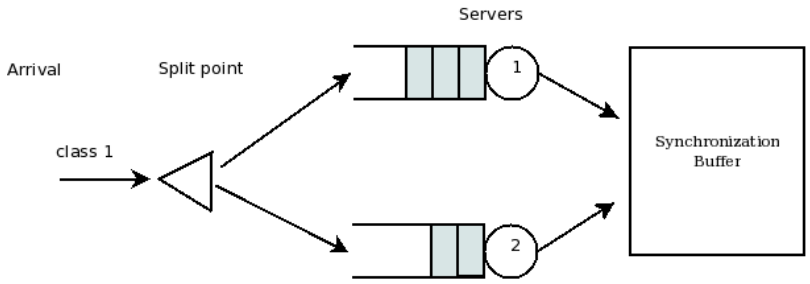


Figure 2.7: System 1 of synchronization time analysis

$$\mathbb{E}[T'_{syn}] = \frac{1}{\gamma_1} + \frac{1}{\gamma_2} - \frac{2}{\gamma_1 + \gamma_2}.$$

In order to obtain $\mu_2^{*(os)}$, we take the derivative of T'_{syn}, and let it equal to 0:

$$\frac{d\mathbb{E}[T'_{syn}]}{d\gamma_2} = -\frac{1}{\gamma_2} + \frac{2}{(\gamma_1 + \gamma_2)^2} = 0.$$

We rewrite it as an expression of $\mu_2^{*(os)}$:

$$\frac{\mu_2^{*(os)} - \lambda}{\mu_1 - \lambda} = 1 + \sqrt{2}.$$

We can observe from Table 2.15 that, the OS approximation is a close approximation for real optimum μ^*, especially in the case of high and low work load. Furthermore, when the server has a high work load, when μ_2 increases, the decrease in sojourn time is smaller than the increase in synchronization time. It is therefore more "beneficial" to take $\mu_2 = \mu_2^{*(os)}$.

System 2 ($N = 1, M = 3$, for fixed μ_1)

In this system (Figure 2.8), we add one more server to the previous system.

ρ_1	μ_1	μ_2^*	$\mu_2^{*(os)}$	$\mathbb{E}T_{syn}(\mu_2^*)$	$\mathbb{E}T_{syn}(\mu_2^{*(os)})$	$\Delta\%$	$\mathbb{E}S(\mu_2)$	$\mathbb{E}S(\infty)$
0.1	10.0	22.0	22.7	0.091	0.091	0.00%	0.124	0.111
0.2	5.0	9.6	10.7	0.203	0.203	0.00%	0.278	0.250
0.3	3.3	5.9	6.7	0.343	0.345	0.58%	0.476	0.429
0.4	2.5	4.2	4.6	0.527	0.531	0.76%	0.739	0.667
0.5	2.0	3.0	3.4	0.777	0.787	1.29%	1.102	1.000
0.6	1.7	2.2	2.6	1.133	1.153	1.80%	1.648	1.500
0.7	1.4	1.7	2.0	1.736	1.773	2.13%	2.565	2.333
0.8	1.3	1.4	1.6	2.929	2.966	2.39%	4.370	4.000
0.9	1.1	1.2	1.3	6.389	6.662	4.27%	9.722	9.000

Table 2.15: Performance analysis of Fork-Join queue with 1 class 2 servers (fixed μ_1)

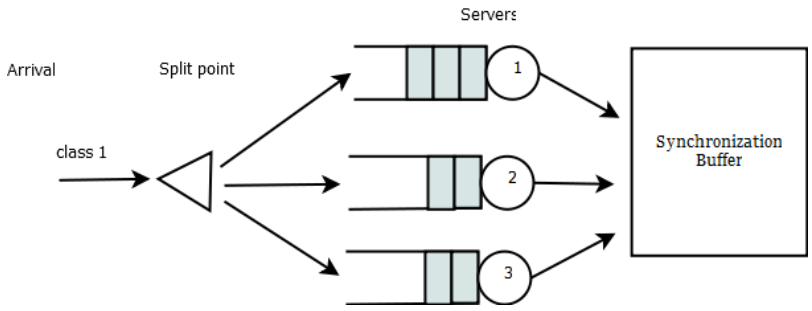


Figure 2.8: System 2 of synchronization time analysis

After applying formula (2.4), we obtain

$$\mathbb{E}[T'_{syn}(\gamma_2,\gamma_3)] = \frac{1}{\gamma_1}+\frac{1}{\gamma_2}+\frac{1}{\gamma_3}-\frac{1}{\gamma_1+\gamma_2}-\frac{1}{\gamma_1+\gamma_3}-\frac{1}{\gamma_2+\gamma_3}.$$

First, we will determine the stationary points of $\mathbb{E}[T'_{syn}]$ as functions of $\gamma_2^{*(os)}$ and $\gamma_3^{*(os)}$.

$$\nabla\mathbb{E}[T'_{syn}] = \begin{pmatrix} -\frac{1}{\gamma_2^{*(os)^2}} + \frac{1}{(\gamma_1+\gamma_2^{*(os)})^2} + \frac{1}{(\gamma_2^{*(os)}+\gamma_3^{*(os)})^2} \\ -\frac{1}{\gamma_3^{*(os)^2}} + \frac{1}{(\gamma_1+\gamma_3^{*(os)})^2} + \frac{1}{(\gamma_2^{*(os)}+\gamma_3^{*(os)})^2} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}. \tag{2.5}$$

Notice that the this system is symmetric with respect to γ_2 and γ_3. It is therefore reasonable to assume $\mu_2^{*(os)} = \mu_3^{*(os)}$. Therefore, by solving the previous equation, we get:

$$\gamma_2^{*(os)} = \gamma_3^{*(os)} = (2\sqrt{3}+3)\gamma_1,$$

rewrite it as an expression of $\mu^{*(os)}$, we get:

$$\frac{\mu^{*(os)}-\lambda}{\mu_1-\lambda} = 2\sqrt{3}+3. \tag{2.6}$$

Now, the next question is: whether formula (2.6) return a global minimum? We will present the following theorem which tell us that equation (2.6) is a strict global optimum.

It is easy to show that this point is a unique stationary point.

Theorem 2.1. *The point $(\mu_2^{*(os)}, \mu_3^{*(os)})$ of $\mathbb{E}[T'_{syn}]$ is the unique stationary point with $\mu_2^{*(os)}, \mu_3^{*(os)} < \infty$ in system 2.*

Proof. We will prove this by contradiction. Suppose that there is another stationary point $(\mu_2^{*(os)}, \mu_3^{*(os)})$, with $\mu_2^{*(os)} \neq \mu_3^{*(os)}$. Without loss of generality we may assume that $\mu_2^{*(os)} = \mu_3^{*(os)} + \delta$, for some $\delta > 0$. This point satisfies equation (2.5). Hence,

$$-\frac{1}{(\gamma_3+\delta)^2} + \frac{1}{(\gamma_1+\gamma_3+\delta)^2} = \frac{1}{(2\gamma_3+\delta)^2} \tag{2.7}$$

$$-\frac{1}{\gamma_3^2} + \frac{1}{(\gamma_1+\gamma_3)^2} = \frac{1}{(2\gamma_3+\delta)^2} \tag{2.8}$$

We define $g(\delta) := -\frac{1}{(\gamma_3+\delta)^2} + \frac{1}{(\gamma_1+\gamma_3+\delta)^2}$. According to previous equations, we must have $g(0) = g(\delta)$.

Take the derivative of $g(\delta)$, we have:

$$g'(\delta) = \frac{1}{2(\gamma_3+\delta)^3} - \frac{1}{2(\gamma_1+\gamma_3+\delta)^3}.$$

Notice that $\gamma_1, \gamma_3 > 0$. Therefore $g'(\delta) > 0, \forall \delta > 0$, and so $g(0) < g(\delta)$. A contradiction.

□

An interesting question is whether this stationary point is a minimizer.

Theorem 2.2. *Let f: $\mathbb{R}^n \to \mathbb{R}$ be in C^2. If $\nabla f(\bar{x}) = 0$ and $\nabla^2 f(\bar{x})$ is positive definite, then the point $\bar{x}$ is a strict local minimizer of function f.*

Noticing that $\nabla^2 \mathbb{E}[T'_{syn}(\mu^{*(os)}, \mu^{*(os)})]$ is positive definite, and $\nabla \mathbb{E}[T'_{syn}(\mu^{*(os)}, \mu^{*(os)})]$ is equal to 0, which implies that $(\mu^{*(os)}, \mu^{*(os)})$ is a strict global minimum.

ρ_1	μ_1	$\mu_2^* = \mu_3^*$	$\mu^{*(os)}$	$\mathbb{E}T_{syn}(\mu_2^*)$	$\mathbb{E}T_{syn}(\mu^{*os})$	$\Delta\%$	$\mathbb{E}S(\mu^{*os})$	$\mathbb{E}S(\infty)$
0.1	10.0	58.0	(59.2, 59.2)	0.107	0.107	0.00%	0.115	0.111
0.2	5.0	25.0	(26.9, 26.9)	0.240	0.241	0.42%	0.259	0.250
0.3	3.3	15.0	(16.1, 16.1)	0.410	0.410	0.00%	0.442	0.429
0.4	2.5	9.5	(10.7, 10.7)	0.637	0.638	0.16%	0.688	0.667
0.5	2.0	6.0	(7.5, 7.5)	0.949	0.955	0.63%	1.030	1.000
0.6	1.7	4.0	(5.3, 5.3)	1.412	1.423	0.78%	1.537	1.500
0.7	1.4	2.9	(3.8, 3.8)	2.178	2.204	1.19%	2.386	2.333
0.8	1.3	1.9	(2.6, 2.6)	3.699	3.763	1.73%	4.086	4.000
0.9	1.1	1.4	(1.7, 1.7)	8.099	8.385	3.20%	9.136	9.000

Table 2.16: Performance analysis of Fork-Join queue with 1 classes 3 servers (fixing μ_1)

An interesting question is does $\mu_2^{*(os)} = \cdots = \mu_M^{*(os)}$ hold for more queues? In the following system we will use a simplex search to search all the grid point in order to find the global minimum. This search method is called *fminsearch* function in Matlab. Surprisingly, the experiments show that the global minimum has property $\mu_2^{*(os)} = \cdots = \mu_M^{*(os)}$.

System 3 ($N = 1, M = 4$, for fixed μ_1)

For the system with 1 class 4 queues (see figure 2.9), the result is presented in the table below (Table 2.17). In this system, we will only start with work load 0.4, because $\mu^{*(os)}$ is already too high for low work load, which leads to inaccuracy in the simulation result.

Again in this system, the *OS* approximation of optimum service rates has property $\mu_2^{*(os)} = \mu_3^{*(os)} = \mu_4^{*(os)}$. We will give the proof that this is a global minimum later after the proof of system 5.

System 4 ($N = 1, M = 3$, for fixed μ_1, μ_2, with $\mu_1 = \mu_2$)

In the previous systems, we fixed the service rate of one server and vary the service rate of other servers. An interesting question to consider is whether *OS* approximation is also useful if we fix service rates of two servers. To study this,

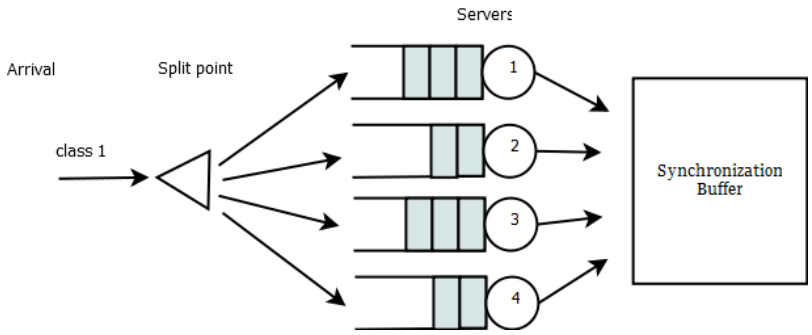


Figure 2.9: System 3 of synchronization time analysis

ρ_1	μ_1	μ^*	$\mu^{*(os)}$	$\mathbb{E}T_{syn}(\mu^*)$	$\mathbb{E}T_{syn}(\mu^{*(os)})$	$\Delta\%$	$\mathbb{E}S(\mu^{*os})$	$\mathbb{E}S(\infty)$
0.4	2.5	18.8	(20.8, 20.8, 20.8)	0.656	0.659	0.46%	0.674	0.667
0.5	2.0	12.7	(14.2, 14.2, 14.2)	0.983	0.986	0.31%	1.012	1.000
0.6	1.7	8.6	(9.8, 9.8, 9.8)	1.465	1.470	0.34%	1.501	1.500
0.7	1.4	5.2	(6.7, 6.7, 6.7)	2.287	2.293	0.44%	2.354	2.333
0.8	1.3	3.5	(4.3, 4.3, 4.3)	3.898	3.923	0.64%	4.033	4.000
0.9	1.1	2.1	(2.5, 2.5, 2.5)	8.759	8.881	1.39%	9.148	9.000

Table 2.17: Performance analysis of Fork-Join queue with 1 classes 4 servers (fixing μ_1)

we first construct the following experiment: in system 4 (see Figure 2.10), we fix $\mu_1 = \mu_2 = 2$, $\lambda = 1$, and we let μ_3 varying from 1.9 to 9. The simulation results with its 95% confidence interval and *OS* approximation result are draw in graph 2.11.

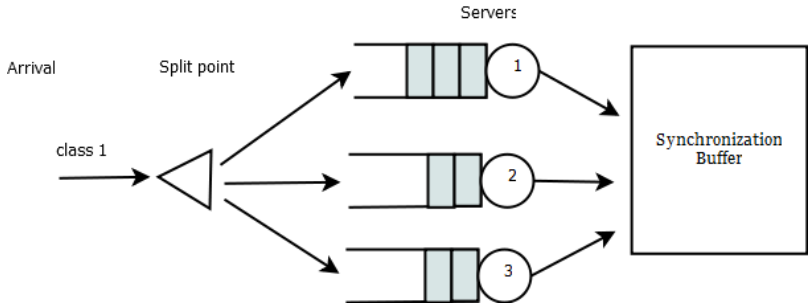


Figure 2.10: System 4 of synchronization time analysis

As one can observe from graph 2.11 that *OS* approximation is not close to the real system in this case. However, both graphs have the same shape with minimizer that are close to each other. Therefore, we still can use the *OS* method to help us find the near minimizer of the synchronization time.

We now evaluate the performance of the *OS* method in system 4 with fixed service rates of the first two servers. We again consider 9 different scenarios with different work load of the first two queues. The performance is shown in table 2.18.

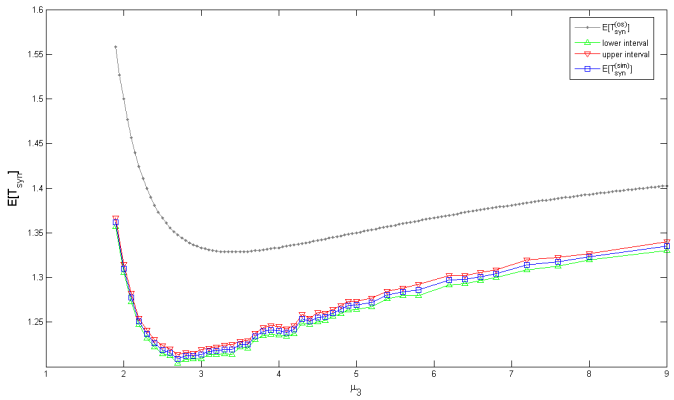


Figure 2.11: Synchronization time of system 4

ρ_1	$\mu_1 = \mu_2$	μ_3^*	$\mu_3^{*(os)}$	$\mathbb{E}T_{syn}(\mu_3^*)$	$\mathbb{E}T_{syn}(\mu_3^{*os})$	$\Delta\%$	$\mathbb{E}S(\mu_3^{*os})$	$\mathbb{E}S(\infty)$
0.1	10.0	22.4	22.7	0.145	0.146	0.69%	0.171	0.165
0.2	5.0	9.6	10.7	0.321	0.322	0.31%	0.381	0.369
0.3	3.3	5.6	6.6	0.540	0.542	0.37%	0.647	0.627
0.4	2.5	3.9	4.6	0.822	0.827	0.61%	0.994	0.967
0.5	2.0	2.8	3.4	1.206	1.218	1.00%	1.475	1.438
0.6	1.7	2.2	2.6	1.765	1.795	1.70%	2.195	2.138
0.7	1.4	1.8	2.0	2.689	2.731	1.56%	3.386	3.295
0.8	1.3	1.5	1.6	4.508	4.592	1.86%	5.729	5.600
0.9	1.1	1.2	1.3	9.740	10.222	4.78%	12.664	12.489

Table 2.18: Performance analysis of Fork-Join queue with 1 classes 3 servers (fixing $\mu_1 = \mu_2$)

As we can see from table 2.18 that the *OS* method still provides a near minimizer of the synchronization time.

System 5 ($N = 1, M = 4$, for fixed μ_1, μ_2)

In system 5 (Figure 2.9), we first will prove that the minimizer of the *OS* approximation has property $\mu_3^{*(os)} = \mu_4^{*(os)}$. Then we verify whether this is also true in the real system. The real system has two scenarios $\mu_1 = \mu_2$ and $\mu_1 \neq \mu_2$. We will present numerical result for both scenarios.

Theorem 2.3. *In system 5 with fixed μ_1 and μ_2, the global minimizer has property $\mu_3^{*(os)} = \mu_4^{*(os)}$.*

Proof. Apply formula (2.4), and take the derivative of $\mathbb{E}[T'_{syn}] = 0$, we get

$$\begin{aligned} -\frac{1}{\gamma_3^2} + \frac{1}{(\gamma_1+\gamma_3)^2} + \frac{1}{(\gamma_2+\gamma_3)^2} - \frac{1}{(\gamma_1+\gamma_2+\gamma_3)^2} = &- \frac{2}{(\gamma_1+\gamma_2+\gamma_3+\gamma_4)^2} \\ &- \frac{1}{(\gamma_4+\gamma_3)^2} + \frac{1}{(\gamma_1+\gamma_3+\gamma_4)^2} \\ &+ \frac{1}{(\gamma_2+\gamma_3+\gamma_4)^2}. \end{aligned} \tag{2.9}$$

$$\begin{aligned} -\frac{1}{\gamma_4^2} + \frac{1}{(\gamma_1+\gamma_4)^2} + \frac{1}{(\gamma_2+\gamma_4)^2} - \frac{1}{(\gamma_1+\gamma_2+\gamma_4)^2} = &- \frac{2}{(\gamma_1+\gamma_2+\gamma_3+\gamma_4)^2} \\ &- \frac{1}{(\gamma_3+\gamma_4)^2} + \frac{1}{(\gamma_1+\gamma_3+\gamma_4)^2} \\ &+ \frac{1}{(\gamma_2+\gamma_3+\gamma_4)^2}. \end{aligned} \tag{2.10}$$

Assume μ_3, μ_4 are the solutions to equations (2.9) and equation (2.10), with $\mu_3 = \mu_4 + \delta, \delta > 0$.

We let $g(\delta) = -\frac{1}{(\gamma_4+\delta)^2} + \frac{1}{(\gamma_1+\gamma_4+\delta)^2} + \frac{1}{(\gamma_2+\gamma_4+\delta)^2} - \frac{1}{(\gamma_1+\gamma_2+\gamma_4+\delta)^2}$. Notice that the right side of the equation (2.9) and equation (2.10) are the same. Therefore, we have

$$g(\delta) = g(0).$$

Now we will prove that $g(\delta) \neq g(0)$ for any $\delta > 0$.

$$g'(\delta) = \frac{1}{2}[\frac{1}{(\gamma_4+\delta)^3} - \frac{1}{(\gamma_1+\gamma_4+\delta)^3} - \frac{1}{(\gamma_2+\gamma_4+\delta)^3} + \frac{1}{(\gamma_1+\gamma_2+\gamma_4+\delta)^3}].$$

If we let $\frac{1}{\gamma_4+\delta} = a$, $\frac{1}{\gamma_1+\gamma_4+\delta} = b$, $\frac{1}{\gamma_2+\gamma_4+\delta} = c$ and $\frac{1}{\gamma_1+\gamma_2+\gamma_4+\delta} = d$, we have:

$$g'(\delta) = \frac{1}{2}[(a-b)(a^2+ab+b^2) - (c-d)(c^2+cd+d^2)] > 0.$$

This is because $(a-b) > (c-d)$ and $(a^2+ab+b^2) > (c^2+cd+d^2)$.

Therefore, we have a contradiction. In conclusion, the global minimizer has property $\mu_3 = \mu_4$.

□

We have proved that the stationary point (μ_3, μ_4) has property $\mu_3 = \mu_4$. Now we will show that there exists at least one such stationary point with $\mu_3, \mu_4 < \infty$.

Apply formula (2.4), and let it equal to 0, we have

$$\begin{aligned} &-\frac{1}{\gamma_3^2} + \frac{1}{(\gamma_1+\gamma_3)^2} + \frac{1}{(\gamma_4+\gamma_3)^2} + \frac{1}{(\gamma_2+\gamma_3)^2} - \frac{1}{(\gamma_1+\gamma_2+\gamma_3)^2} \\ &- \frac{1}{(\gamma_1+\gamma_3+\gamma_4)^2} - \frac{1}{(\gamma_2+\gamma_3+\gamma_4)^2} + \frac{2}{(\gamma_1+\gamma_2+\gamma_3+\gamma_4)^2} = 0. \end{aligned} \tag{2.11}$$

By theorem 2.3, know that $\gamma_3 = \gamma_4$. Therefore, equation (2.11) can also be written as

$$\begin{aligned} f(\gamma_3) = &-\frac{3}{4\gamma_3^2} - \frac{1}{(\gamma_1+\gamma_2+\gamma_3)^2} - \frac{1}{(\gamma_2+\gamma_3+\gamma_4)^2} - \frac{1}{(\gamma_1+\gamma_3+\gamma_4)^2} \\ &+ \frac{1}{(\gamma_1+\gamma_3)^2} + \frac{1}{(\gamma_2+\gamma_3)^2} + \frac{2}{(\gamma_1+\gamma_2+\gamma_3+\gamma_4)^2} = 0. \end{aligned} \tag{2.12}$$

If $\gamma_3 = 0$, $f(\gamma_3) = -\infty$. If γ_3 is very large, then $-\frac{3}{4\gamma_3^2}$ and $\frac{1}{(\gamma_1+\gamma_3)^2}$ are the dominating terms, with $\frac{1}{(\gamma_1+\gamma_3)^2} - \frac{3}{4\gamma_3^2} > 0$. This means $f(\gamma_3) > 0$, when γ_3 is large. Therefore, there must exist $\gamma_3^* < \infty$, such that $f(\gamma_3^*) = 0$, and $\mu_3 = \mu_4 = \gamma_3^* - 1$ is the stationary point.

Remark. *Using this theorem, we can extend the property to system 5 with fixing only μ_1. Specifically, in Figure 2.9 with fixed μ_1, assume $\mu_2^{*(os)} \neq \mu_3^{*(os)} \neq \mu_4^{*(os)}$. According to theorem 2.3, we can further decrease the expected synchronization time by setting $\mu_3 = \mu_4$. The same argument also leads to $\mu_2 = \mu_3$. Therefore, $\mu_2^{*(os)} = \mu_3^{*(os)} = \mu_4^{*(os)}$ should hold .*

$\mu_1 = \mu_2$

Now we consider the system 5 with fixed service rates ($\mu_1 = \mu_2$) of the first two servers. $\mu_3^{*(os)}$ and $\mu_4^{*(os)}$ are obtained by *fminsearch*. The performance is shown in table 2.19.

We can see that the *OS* method return a value which is close to minimizer of the real system. Furthermore, it appears that the minimizer has the property $\mu_3^{*(os)} = \mu_4^{*(os)}$ in all scenarios.

ρ_1	$\mu_1 = \mu_2$	(μ_3^*, μ_4^*)	$(\mu_3^{*os}, \mu_4^{*os})$	$\mathbb{E}T_{syn}(\mu^*)$	$\mathbb{E}T_{syn}(\mu^{*os})$	$\Delta\%$	$\mathbb{E}S(\mu^{*os})$	$\mathbb{E}S(\infty)$
0.1	10.0	(33.7, 33.7)	(35.6, 35.6)	0.157	0.158	0.64%	0.169	0.165
0.2	5.0	(15.4, 15.4)	(16.4, 16.4)	0.348	0.348	0.00%	0.375	0.369
0.3	3.3	(8.9, 8.9)	(10.0, 10.0)	0.589	0.592	0.51%	0.639	0.627
0.4	2.5	(5.8, 5.8)	(6.8, 6.8)	0.907	0.911	0.44%	0.985	0.967
0.5	2.0	(4.3, 4.3)	(4.8, 4.8)	1.342	1.355	0.97%	1.471	1.438
0.6	1.7	(2.6, 2.6)	(3.6, 3.6)	1.982	2.003	1.06%	2.167	2.138
0.7	1.4	(2.0, 2.0)	(2.7, 2.7)	3.027	3.077	1.65%	3.364	3.295
0.8	1.3	(1.5, 1.5)	(2.0, 2.0)	5.067	5.196	2.55%	5.714	5.600
0.9	1.1	(1.3, 1.3)	(1.4, 1.4)	11.135	11.787	5.86%	12.622	12.489

Table 2.19: Performance analysis of Fork-Join queue with 1 classes 4 servers ($\mu_1 = \mu_2$)

$\mu_1 \neq \mu_2$

In this case, we will verify whether $\mu_3^* = \mu_4^*$ holds while fixing μ_1 and μ_2, with $\mu_1 \neq \mu_2$. We construct one experiment with parameters $\rho_1 = 0.5, \rho_2 = 0.8$. They stand for "High" and "Medium" work loads. We did not do numerical experiment with low work load. This is because it leads to inaccuracy of the simulation result. For example, the expected synchronization time of experiment with parameters $\rho_1 = 0.2, \rho_2 = 0.8, \rho_3 = 0.4, \rho_4 = 0.5$ is almost equal to the expected synchronization time of experiment with parameters $\rho_1 = 0.2, \rho_2 = 0.8, \rho_3 = 0.5, \rho_4 = 0.5$. The comparison of the *OS* approximation and the simulation result is shown in Table 2.20. The detailed numerical result is shown in Table 2.21.

ρ_1	(μ_3^*, μ_4^*)	$(\mu_3^{*os}, \mu_4^{*os})$	$\mathbb{E}T_{syn}(\mu^*)$	$\mathbb{E}T_{syn}(\mu^{*os})$	$\Delta\%$
(0.5, 0.8)	(3, 3)	(2.9, 2.9)	3.981	3.983	0.05%

Table 2.20: Performance analysis of Fork-Join queue with 1 classes 4 servers ($\mu_1 = 2, \mu_2 = 1.25$)

μ_3 \ μ_4	1.5	2.0	3.0	4.0	5.0	6.0	7.0
1.5	4.264	4.129	4.232	4.335	4.375	4.407	4.431
2.0	4.134	3.915	3.987	4.054	4.046	4.099	4.102
3.0	4.230	3.987	3.981	3.997	4.008	4.028	4.053
4.0	4.300	4.027	3.995	4.019	4.014	4.039	4.054
5.0	4.366	4.081	4.022	4.025	4.013	4.012	4.041
6.0	4.416	4.114	4.027	4.043	4.056	4.071	4.060
7.0	4.408	4.123	4.057	4.071	4.054	4.055	4.052

Table 2.21: Expected synchronization time of Fork-Join queue with 1 classes 4 servers (fixing $\mu_1 = 2, \mu_2 = 1.25$)

As we can observe from Table 2.21 that the real minimum is achieved by taking $\mu_3 = \mu_4$. This coincide with the *OS* approximation, where $\mathbb{E}[T'_{syn}]$ is achieved by taking $\mu_3^{*(os)} = \mu_4^{*(os)}$.

More systems ($N = 1, M$, for fixed $\mu_1, \mu_2, \dots \mu_i$)

For systems with more than four servers, we can not prove whether $\mu_{i+1}^{*(os)} = \mu_{i+2}^{*(os)} = \cdots = \mu_M^{*(os)}$ holds. We state that it is only necessary to prove $\mu_{M-1}^{*(os)} = \mu_M^{*(os)}$ holds for systems with fixed parameters $\mu_1, \mu_2, \dots \mu_{M-2}$. Despite the proof, according to the results for systems showed previously, we formulate the following conjecture.

Conjecture. *In the OS approximation, if we fixed the service rates of i servers ($i \geq 1$), the minimum expected synchronization time is achieved by taking other service rates to be equal. This holds not only for the OS approximation, but also for the real system.*

2.2.3 Conclusion

In the previous subsection, we showed that the service rates play an important role in synchronization buffer size optimization. This is also the reason why we consider the synchronization time. We developed the *OS* approximation method to approximate the synchronization time in order to helps us to allocate the service rates, at which the minimum synchronization time is achieved. Through numerical results of several systems, we see that this method provides both close approximation to real system synchronization time and a close allocation of the minimizer. However, this does not hold for all systems. In system 4 of this section, we have seen that the *OS* method performs badly in approximation of the synchronization time. However, this does not handicap that the *OS* method provides a close allocation of the minimizer that reduce the real system synchronization time. It is intuitively clear that less synchronization time leads to smaller synchronization buffer size.

According to the experiments, another interesting fact is that the optimal service rates of non-fixed servers are larger when less servers have fixed rates.

The systems we show include up till to 4 servers. Whether there exists a minimum synchronization time in system with more servers still need to be investigated.

In all the systems in this thesis, it is surprising that if we fix service rates for one or two servers, using the *OS* approximation, the minimum synchronization time is achieved by setting other service rates to be equal. We gave the proof that for system with one fixed service rate and two unfixed service rates, the strict global minimizer has property $\mu_2^{*(os)} = \mu_3^{*(os)}$. We also gave the proof that for system with two fixed service rates and two unfixed service rates, the strict global minimizer has property $\mu_3^{*(os)} = \mu_4^{*(os)}$. Furthermore, we verified that such properties also hold in real systems.

Synchronization time vs. Sojourn time

Now the question is that whether it is beneficial to lessen the synchronization time while raise the sojourn time. Clearly, there is a trade off. We let $\beta = \frac{|\mathbb{E}[T_{syn}(\mu^{*(os)})] - \mathbb{E}[T_{syn}(\infty)]|}{|\mathbb{E}[S(\infty)] - \mathbb{E}[S(\mu^{*(os)})]|}$, which stands for the ratio of the increases in synchronization time and the decreases in sojourn time when we comparing $\mu = \mu^{*(os)}$ and $\mu = \infty$. We observe from all the numerical results that we have $\beta \geq 1$.

Furthermore, if the fixed server has a high work load, the system has a higher β. This means that in a situation of high work load, it is more "beneficial" to take $\mu = \mu^{*(os)}$. In conclusion, the *OS* method offers us a way of reconstructing the system to reduce buffer size without much increment of the sojourn time. This is especially true if the fixed servers have a high work load.

In this thesis, we only consider the synchronization time of systems with one job class. However, buffer size optimization has a larger significance in a multi-class systems. This is because the buffer size will expand when all classes wait in the buffer. Therefore, the *OS* method will offer more improvement to buffer optimization in multi-class system.

2.3 Future work

In this section, we will discuss some questions that have not been answered in this thesis.

First of all, it would be interesting to evaluate the performance of the *MOS* method in more complicated systems, e.g. more customer classes or more servers for each class. If the *MOS* method approximate badly in systems with more servers, other methods of approximation are also of interest. Secondly, as mentioned in the conjecture, we think that *MOS* works well in the case of general service time distributions, e.g. the Erlang distribution. It is important to validate this conjecture, since other type of service time distributions arise regularly in practice.

Regarding theoretical aspects of the sojourn time, we are eager to prove or disprove that the *MOS* method provides an upper bound for the sojourn time of the real system.

For the synchronization time analysis, we used the *OS* method to find the service rates that minimize synchronization time in a system with less than five servers. The case with more than five servers for each class remains an open problem.

As we presented in the numerical results, using the *OS* method, the minimum expected synchronization time is achieved by set unfixed service rates to be equal for systems with at most four servers. Whether this is true for system with more servers needs further investigation.

The numerical results show that the approximating model with independent arrivals provides an upper bound. It will be interesting to study whether this approximation provides an upper bound for all systems.

As mentioned before, formula (2.4) does not return the synchronization time of the independent system. Therefore, it is interesting to compare the independent system and the original system in the "synchronization" time. Here we formulate the synchronization time in terms of the stationary distribution. To illustrate this we consider a system with two servers. The dependent system is shown is Figure 2.12 and the corresponding independent system is shown in Figure 2.13.

In system I, we assume the system is a positive recurrent Markov process. Then the stationary distribution $\{\pi_{i_1,i_2,i_3,i_4}\}_{i\in S}$ exist. Here i_1 stands for the number

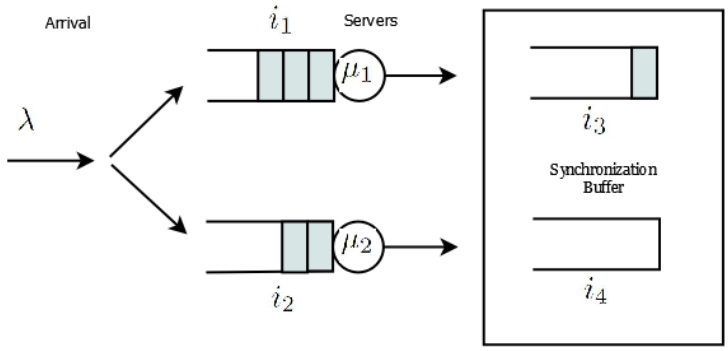


Figure 2.12: System I

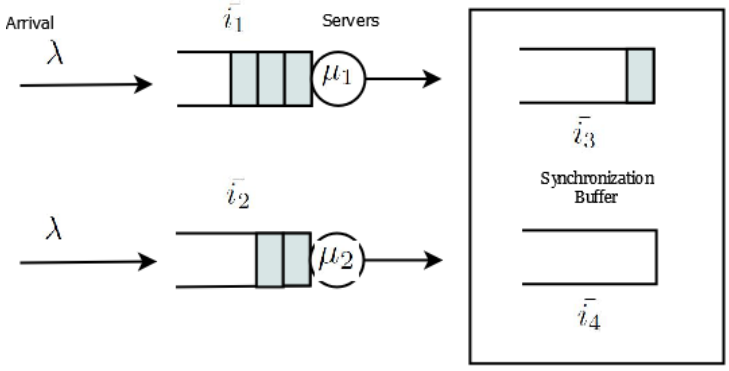


Figure 2.13: System II

of sub-jobs in the first queue plus the sub-job at the first server, i_2 stands for the number of sub-jobs in the second queue plus the sub-job at the second server, i_3 stands for the number of sub-jobs in the synchronization buffer after having been served by the first server, i_4 stands for the number of sub-jobs in the synchronization buffer after having been served by the second server, $i = (i_1, i_2, i_3, i_4)$, S is the state space. Clearly, in system I, $S = \{(i_1, i_2, i_3, i_4)|i_j = \mathbb{Z}_+, \forall j,\ i_3 \cdot i_4 = 0, i_1 + i_3 = i_2 + i_4\}$. It is well-known that π_{i_1,i_2,i_3,i_4} can be interpreted as the fraction of time the system spends in state i. As a consequence, the synchronization time T^I_{syn} of system I can be expressed in the following way:

$$T^I_{syn} = \sum_i \pi_{i_1,i_2,i_3,i_4}(i_3 + i_4). \tag{2.13}$$

The notation in system II is similar. We denote by $\bar{i_1}$ the number of sub-jobs in the first queue plus the sub-job in the first server, by $\bar{i_2}$ the number of sub-jobs in the second queue plus the sub-job in the second server, by $\bar{i_3}$ the number of sub-jobs in the synchronization buffer after having been served by the first server, by $\bar{i_4}$ the number of sub-jobs in the synchronization buffer after having been served by the second server, $\bar{i} = (\bar{i_1}, \bar{i_2}, \bar{i_3}, \bar{i_4})$, $\bar{S}$ is the state space with $\bar{S} = \{(\bar{i_1}, \bar{i_2}, \bar{i_3}, \bar{i_4})|\bar{i_j} \in \mathbb{Z}_+, \forall j,\ \bar{i_3} \cdot \bar{i_4} = 0\}$. System II has stationary distribution $\{\bar{\pi}_{\bar{i_1},\bar{i_2},\bar{i_3},\bar{i_4}}\}_{\bar{i} \in \bar{S}}$. The "synchronization" time of system II can be written as:

$$T^{II}_{syn} = \sum_{\bar{i}} \bar{\pi}_{\bar{i_1},\bar{i_2},\bar{i_3},\bar{i_4}}(\bar{i_3} + \bar{i_4}). \tag{2.14}$$

Comparison and computation of formula (2.13) and formula (2.14) are essential in analyzing the synchronization time of Fork-Join queueing systems. Furthermore, formula (2.14) might be another approach for approximating the expected synchronization time of the original system. We are also interested in how formula (2.14) related to *OS* approximation. We will leave these for future work.

Part II

Stochastic knapsack problem

Chapter 3

Introduction to stochastic knapsack problem

3.1 Background

We will now consider another problem called the stochastic knapsack problem. This model was first introduced by Ross and Tsang in [14]. It is derived from the famous knapsack problem, except for the fact that in this model the arrivals processes are stochastic processes. This type of model has many applications. We will give few examples.

Example 1: In a mobile cellular system, a signal transmission base with limited bandwidth to handle bandwidth request (i.e. phone call, cellular internet browse) in its control area. Normally, each type of request has different properties, for example, initiating a phone call has priority but gives little reward (charges) to the system. On the other hand, internet users have less priority, it requests more bandwidth than a phone call but gives more reward (charges) to the system. Therefore, we are motivated to design a policy that maximizes the reward for such a system while handling all different kinds of request.

Example 2: In a circuit-switched telecommunication system, a central server supports a variety of traffic types (i.e. video, image, voice, etc). Each type of traffic has a different bandwidth request and holding time distribution. The problem of optimally accepting calls in order to maximize average revenue is equivalent to the stochastic knapsack problem [14].

The stochastic knapsack problem was first studied in [14]. In [14], the authors consider the MaxAcR and MinBlC criteria and derive the optimal *coordinate convex* policy for this problem. This type of policy leads to a product-form steady state distribution. Ross and Yao study the effect of changed parameter on this policy in [15]. In [13], Ramjee, Towsley and Nagarajan extend the problem to other objective functions and derive the optimal policy for each objective function as well as some monotonicity properties. Feinberg and Reiman [4] study the special case where the service rates and rewards do not depend on the customer class. A fluid model approximation method is developed in [1]. In [11] and [12],

two policies called threshold policy and reservation policy are studied.

3.2 Model formulation and problem description

Traffic from N classes shares B resources. Class i traffic arrives according to a Poisson process with parameter λ_i, $i = 1, \ldots, N$. Each class i customer demands b_i resources, with b_i integer. After an exponentially distributed time with parameter μ_i, these b_i resources will be released simultaneously, and the customer leaves the system. The state of the system is denoted by $x = (x_1, \ldots, x_N)$, where x_i stands for the number of class i customers in the system. There is no waiting room, i.e. customers who do not find enough resources upon arrival, are rejected automatically. Besides this, upon arrival of a customer, the system can either accept or reject him according to a pre-determined *Call Admission Control (CAC)* policy. A simultaneous reward r_i is earned upon acceptance of a class i customer (see Figure 3.1). We consider the total discounted reward criterion with discount factor $\beta \in (0, 1)$. This means that gaining reward r at time t is worth $r\beta^t$ now. The objective is to find a *CAC* policy, that optimizes the *Quality of Service (QoS)*.

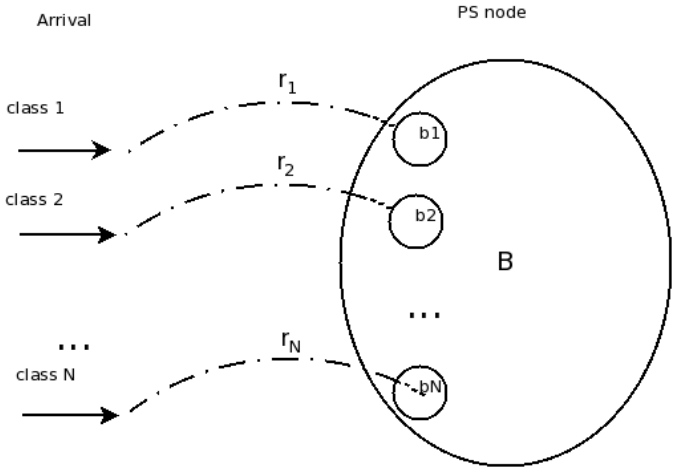


Figure 3.1: Example of Stochastic Knapsack Problem

In [13], the QoS is measured in 3 different ways:

1. **MaxAcR:** Maximize the average reward of the system over an infinite time horizon.

2. **MinBlC:** Instead of gaining rewards by accepting customers, it is also possible to measure QoS by a penalty cost for rejecting customers. In this way, MinBlC stands for minimizing the blocking costs.

3. **MinBLOCK:** For a given number of channels, this criterion minimizes the overall blocking probability subject to the constraint that the blocking probability of a certain type of customer may not exceed a given threshold.

In this thesis, we only investigate MaxAcR and MinBlC. As a matter of fact,

MaxAcR and MinBlC are equivalent ways of measuring QoS. This is directly deduced from the objective functions.

MaxAcR	MinBlC
$R^* = \max \sum_i r_i \cdot (1 - B_i) \cdot \lambda_i,$	$R^* = \min \sum_i c_i \cdot B_i \cdot \lambda_i$

where B_i is the blocking probability for the ith class customer.

3.3 Goal and structure

The objective of this part is to find a policy that maximizes the reward of the system. We will present how we derive optimal policy. Furthermore, we will compare two other policies which perform close to optimal with less computation complexity. Although those methods are well developed, we will show the defects of those policies through some examples.

This part is structured in the following way. In chapter 4, different types of policies are presented. In section 4.1, we show the explicit expression of stationary distribution of complete sharing policy. In section 4.2, we formulate the algorithm to derive the optimal policy by using Markov decision process. Threshold policies and reservation policies are discussed in section 4.3 and 4.4, respectively. The performances of these policies are evaluated in section 4.5. The future work is discussed in section 4.7.

Chapter 4

Analysis of policies

In this chapter, we will investigate four different types of policies for the stochastic knapsack problem. First, we show how to derive such policies, and then compare their performance. The four types of CAC policies studied are:

Complete Sharing policy
A customer is accepted whenever the system has sufficient resources, otherwise he is blocked.

Optimal policy
The policy that returns the maximum long term average reward to the system.

Reservation policies
Under a reservation policy, a class i customer is accepted if and only if the number of occupied resources do not exceed a reservation parameter c_i ($c_i \leq B$) after acceptance. Formally, a class i customer is accepted if and only if $\sum_{j=1}^{N} x_j \cdot b_j + b_i \leq c_i$.

Threshold policies
Under a threshold policy, a class i customer is accepted if and only if there are available resources in the system and the number of class i customer does not exceed a given threshold t_i after acceptance. Specifically, a class i customer is accepted if and only if $x_i + 1 \leq t_i$ and $\sum_{j=1}^{N} x_j \cdot b_j + b_i \leq B$.

4.1 Complete sharing policy

The simplest policy is complete sharing policy. The system is well-known as the Multi-Rate Model under a complete sharing policy. A complete sharing policy leads to an aperiodic and irreducible Markov process [14] with product-form stationary distribution [14][11]. The steady state distribution is given by [11].

$$\pi(x) = \pi(x_1, x_2, \ldots, x_N) = \frac{1}{G} \prod_{i=1}^{N} \frac{\rho_i^{x_i}}{x_i!}, \forall x \in \Omega,$$

where Ω is the state space, $\rho_i = \frac{\lambda_i}{\mu_i}$, and $G := \sum_{x\in\Omega} \prod_{i=1}^{N} \frac{\rho_i^{x_i}}{x_i!}$ the normalizing constant.

We denote by Ω_i ($\Omega_i \subseteq \Omega$) the set of states in which a class i customer will be accepted. The blocking probability can be expressed in the following way:

$$B_i = 1 - \sum_{x\in\Omega_i} \pi(x) = 1 - \frac{\sum_{x\in\Omega_i} \prod_{j=1}^{N} \rho_j^{x_j}/x_j!}{\sum_{x\in\Omega} \prod_{j=1}^{N} \rho_j^{x_j}/x_j!}. \tag{4.1}$$

Having calculated the blocking probability for each customer class, we can calculate the long-term average reward R under this policy. However, we need to mention that although this formula is exact, in the case of a large state space, a numerical overflow problem may occur when calculating the normalizing constant G. Therefore, in the case of a large state space, one can apply Kaufman - Roberts Recursion [7] to avoid computational problems.

4.2 Optimal policy

In this subsection, we introduce *Markov Decision Processes (MDP)* to derive the optimal policy for this model. First we will describe the so-called value iteration algorithm for discrete time MDP. To this end, we need to apply a uniformization technique to reduce our continuous time model to discrete time. Then we will compare the performance of optimal to complete sharing policy with respect to the long term average reward R.

4.2.1 Uniformization technique

In each state x, we add a "dummy" transition $r_d(x)$ from x to itself, such that all the states get the same transition rate, τ say. τ should satisfy $R_{out}(x) \leq \tau, \forall x \in \Omega$, where $R_{out}(x)$ stands for the rate out of state x. Thus, in this thesis, we set $\tau = \sum_{i=1}^{N} \lambda_i + \sum_{i=1}^{N} \lfloor B/b_i \rfloor \cdot \mu_i$.

4.2.2 Value iteration algorithm

In MDP, a dynamic programming decision is made in each state to decide whether it is beneficial to accept an arriving customer or not. The value iteration scheme can be formally described as $V_{n+1}(x) = T \cdot V_n(x), \forall x \in \Omega$, where Ω is the state space, and $V_n(x)$ stands for the maximum total reward in state x over a time horizon of length n. In this model, the event operator T is a composition of 3 different types: an arrival operator T_{A_i}, a departure operator $T_{D_i}^{(k)}$, and a constant operator I_c. It is a result from Dynamic programming that for our model $V_n(x) - V_n(0) \to V(x) - V(0)$, where V is the average reward value function.

Arrival Operator

Stands for an arrival of a class i ($i = 1, \ldots, N$) customer, T_{A_i} is defined by:

$$T_{A_i} = \max\{r_i + V_n(x+e_i), V_n(x)\};$$

Departure Operator

Stands for departure of one of the k class i customers, ($i = 1,\ldots,N$ and $k = 1,\ldots,\lfloor B/b_i \rfloor$). it is defined by:

$$T_{D_i}^{(k)} = \begin{cases} V_n(x-e_i), & \text{if } x_i \leq k; \\ V_n(x), & \text{otherwise.} \end{cases}$$

Constant Operator

This operator is an indicator function, which is used to avoid system putting transitions leading out of Ω, i.e. to avoid $\sum_{i=1}^{N} b_i x_i > B$; therefore, we define:

$$I_c(x) = \begin{cases} 0, & \text{if } \sum_{i=1}^{N} b_i x_i \leq B; \\ -\infty, & \text{otherwise.} \end{cases}$$

Using the definition above, the value iteration algorithm for MDP applied to our problem has the following form:

$$\begin{aligned} V_{n+1}(x) &= TV_n(x) \\ &= I_c V_n(x) + \beta \sum_{i=1}^{N} \lambda_i T_{A_i} V_n(x) + \beta \sum_{i=1}^{N} \mu_i \sum_{k=1}^{\lfloor B/b_i \rfloor} T_{D_i}^{(k)} V_n(x) \\ &+ \beta \cdot r_d(x) \cdot V_n(x) \end{aligned}$$

4.3 Threshold policy

In order to develop a policy with amenable structure, it is necessary to find a method to order the customer classes, such that when a low-class customer can be accepted, then a high-class should also be accepted whenever there are enough resources. Here we introduce the term *revenue coefficient* $\alpha_i = \frac{r_i \cdot \mu_i}{b_i}, i = 1, 2, \ldots, N$ [12]. One interpretation of α_i is the reward gained by using one unit of resource per unit time to serve a class i customer. The customer class with a higher revenue coefficient is considered to be more beneficial, hence we give it a higher priority.

In this section we discuss the general idea of deriving a threshold policy. We will not evaluate this method. For those readers who are interested, we refer to [11] and [12].

Since threshold policies lead to a time reversible Markov chain, the steady state distribution has a product-form ([14]) so that the blocking probability can be calculated explicitly by (4.1). However, a direct calculation requires enumerating all states. This is not efficient for large state space cases. In [16], a convolution algorithm has been proposed to evaluate system performance under any threshold policy. This convolution algorithm is more efficient and has computational complexity of $O(B^2 N \log N)$.

So far, we can evaluate the performance of a given threshold policy. The next task is to find an efficient way to search for the optimal threshold policy. Intuitively, brutal-force search (search all possible threshold policies) is feasible but a slow search method. **ICSA_THD** is a searching algorithm proposed in [11] which is a faster method. The experiments in [12] show that the policy obtained from **ICSA_THD** search coincides with the optimal threshold policy, even though **ICSA_THD** is a local search rather than a global search method.

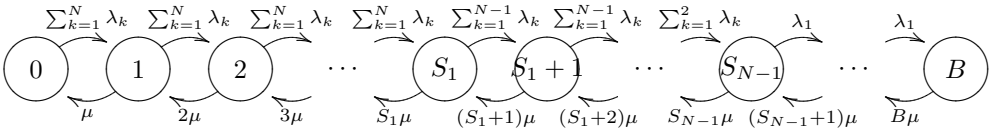
4.4 Reservation policy

In this section, we discuss the general idea of reservation policy without a detailed performance evaluation. In general, reservation policy does not lead to a product-form stationary distribution. Consider the example: $B = 5, b_1 = 1, b_2 = 2, \mu_1 = \mu_2 = 1, r_1 = r_2 = 1$, if we set reservation parameter as $c_1 = 5, c_2 = 3$, then there is no transition rate from state $x = (2, 0)$ to state $y = (2, 1)$. However, there is transition from y to x, because departures can not be blocked. We will show that in the one-dimensional case a reservation policy leads to a product-form stationary distribution. In multi-dimensional cases, we will present a scaling technique that converts the system to one-dimension. Then we translate the optimal reservation policy of one-dimensional system to the original system. Two examples are shown to illustrate that the returned reservation policy can perform badly in the original system.

4.4.1 One-dimensional case

We will consider a special case of the stochastic knapsack model, where all classes have the same resource requirements ($b_i = b, i = 1, 2, \ldots, N$, b can even be equal to 1, without loss of generality) and mean service times ($\mu_i = \mu, i = 1, 2, \ldots, N$). The reason why we call it the *one-dimensional case* is that it can be formulated as a one-dimensional birth-death process. We will derive analytical results for this special case under a reservation policy.

We first order the classes as $1, 2, \ldots, N$ in a decreasing order of α. Let $(B, S_{N-1}, \ldots, S_1)$ denotes the reservation parameters for class $(1, 2, \ldots, N)$ respectively. The system under such policy can be described as a diagram below:



If we let π_i denote the stationary probability of state i, by applying global balance equation, we have:

$$
\begin{cases}
\pi_i = \pi_0(\frac{\sum_{k=1}^{N}\lambda_k}{\mu})^i\frac{1}{i!} & \forall i = 1,2,\ldots,S_1;\\
\pi_i = \pi_{S_1}(\dfrac{\sum_{k=1}^{N-1}\lambda_k}{\mu})^i\dfrac{1}{i!} & \\
\quad = \pi_0(\dfrac{\sum_{k=1}^{N}\lambda_k}{\mu})^{S_1}\dfrac{1}{S_1!}(\dfrac{\sum_{k=1}^{N-1}\lambda_k}{\mu})^i\dfrac{1}{i!} & \forall i = S_1+1, S_1+2,\ldots,S_2;\\
\vdots & \\
\pi_i = \pi_{j-1}(\dfrac{\sum_{k=1}^{N-j+1}\lambda_i}{\mu})^i\dfrac{1}{i!} & \\
\quad = \pi_0(\dfrac{\sum_{k=1}^{N}\lambda_k}{\mu})^{S_1}\dfrac{1}{S_1!}\cdots(\dfrac{\sum_{k=1}^{N-j+1}\lambda_k}{\mu})^i\dfrac{1}{i!} & \forall i = S_{j-1}+1, S_{j-1}+2,\ldots,S_j;\\
\vdots & \\
\pi_i = \pi_0(\frac{\sum_{k=1}^{N}\lambda_k}{\mu})^{S_1}\frac{1}{S_1!}(\frac{\sum_{k=1}^{N-1}\lambda_k}{\mu})^{S_2}\frac{1}{S_2!}\cdots(\frac{\lambda_1}{\mu})^i\frac{1}{i!} & \forall i = S_{N-1}+1, S_{N-1}+2,\ldots,B.
\end{cases}
$$

By using the above equations and the fact that $\pi_0 + \pi_1 + \cdots + \pi_B = 1$, we can evaluate the performance of any given reservation policy.

Now the question is: *How to find an optimal or nearly optimal reservation policy for an one-dimensional problem?* In [11], the authors propose a algorithm called **ICSA_RSV**, which returns a local optimal reservation policy for the one-dimensional problem. In [12], more research has been done about the complexity, convergence and quality of the **ICSA_RSV** algorithm. In a number of cases, **ICSA_RSV** turned our to coincide with the optimal policy.

4.4.2 Multi-dimensional case

As opposed to the threshold and complete sharing policies, a multi-dimensional system under reservation policy is not a time reversible Markov chain. Therefore, it does not lead to a product-form stationary distribution. This property leads us to the thought: *if we can convert all problems into a one-dimensional problem, then we would have an easy way to calculate the blocking probability under the reservation policy.* In [11], the authors propose a scaling technique where they reduce the multi-dimensional case to the one-dimensional equivalent discussed in the previous subsection. Consider class-k customers: there are λ_k arrivals every time unit, each arrival requires b_k resources. In the converted system, we let $\lambda_i' = \frac{\lambda_i b_i}{\mu_i}$, and $b_i = 1, \mu_i = 1, \forall i = 1,2,\ldots,N$ (see Figure 4.1).

After scaling, we can then apply the **ICSA_RSV** algorithm to get the local optimal reservation policy.

4.5 Numerical results

In this section we first study the structure of the optimal policy through some examples. The optimal policy does not always have an amenable structure. We illustrate this by two examples (Table 4.2). Then we do experimental analysis

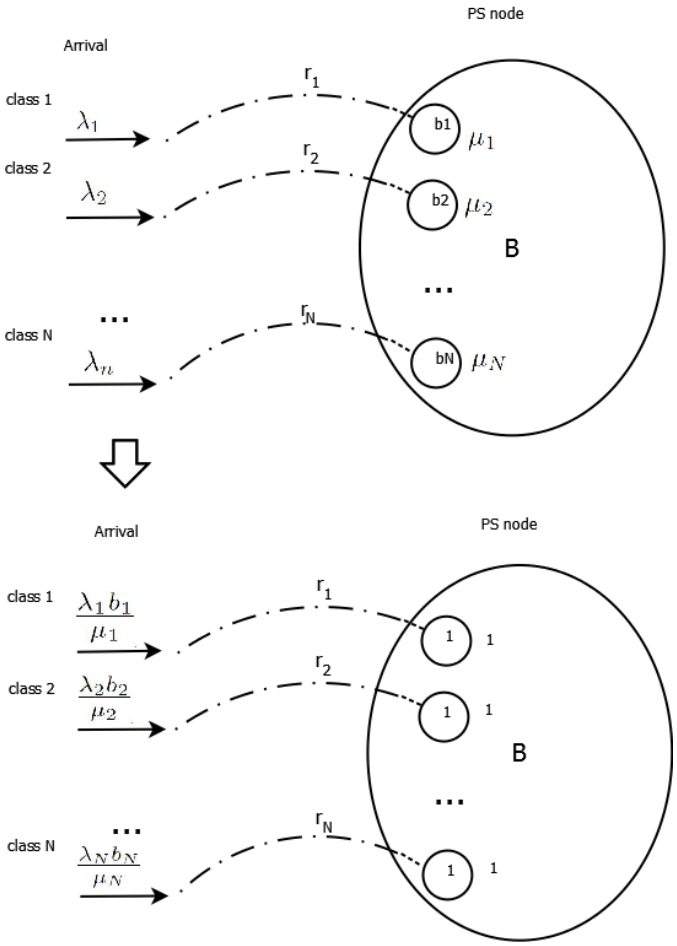


Figure 4.1: Conversion from multi-dimensional case to one-dimensional case

on reservation policy and threshold policy. The experiments will show that the reservation policy can perform really bad. The detailed evaluation of threshold policy and reservation policy can be found in [11].

In the first four examples (Table 4.1), we fix $\beta = 1$. We compare the performance of complete sharing policy to the optimal policy. The experiments are constructed in the following way. We only consider the case where $\lambda > \mu$. This is because when $\lambda < \mu$, the system will almost always accept the arrival. Therefore, in parameter set 1, we take $\rho_1 = 10$, and $\rho_2 = 2$, $\rho = \frac{\lambda}{\mu}$. We variate all different combinations of b and r. The label of analysis is in Table 4.2. All our examples have only two customer classes, in order to have a convenient graphical presentation.

Note that in example 1, the complete sharing policy performs much worse than the optimal policy. This is because class 2 customers are much more beneficial than class 1 customers in this example. Therefore, as we can observe in Figure 4.3, the optimal policy does not allow any acceptance of class 1 customer. In contrast, in the complete sharing policy, a class 1 customer is always allowed as long as there are enough resources.

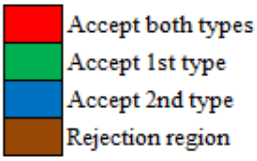


Figure 4.2: Label

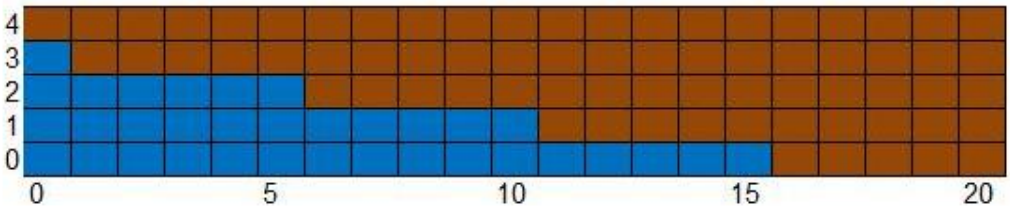


Figure 4.3: Optimal policy of example 1

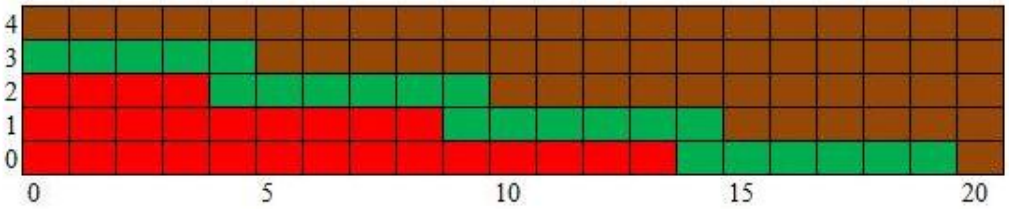


Figure 4.4: Optimal policy of example 2

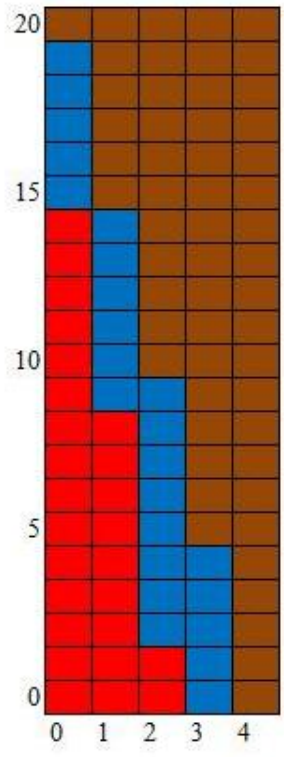


Figure 4.5: Optimal policy of example 3

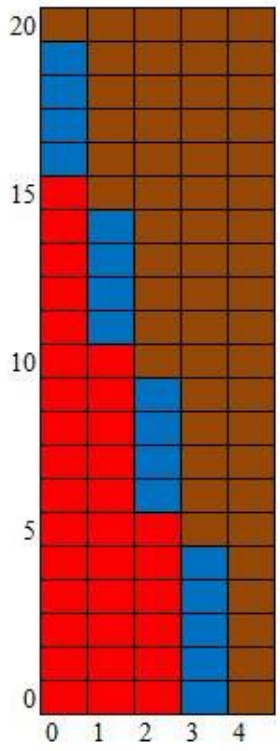


Figure 4.6: Optimal policy of example 4

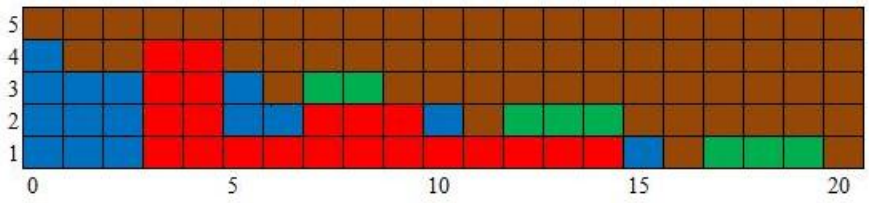


Figure 4.7: Optimal policy of example 5

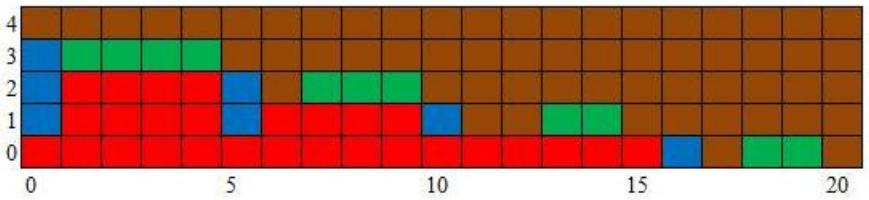


Figure 4.8: Optimal policy of example 6

B	ρ	μ	r	b	R_{Opt}	R_{CS}	Example
20	(10, 2)	(1, 5)	(1, 5)	(1, 5)	45.24	10.29	example 1
20	(10, 2)	(1, 5)	(5, 1)	(1, 5)	53.09	50.94	example 2
20	(10, 2)	(1, 5)	(1, 5)	(5, 1)	51.93	41.85	example 3
20	(10, 2)	(1, 5)	(5, 1)	(5, 1)	23.11	22.49	example 4

Table 4.1: Parameters set 1 for Optimal policy

B	β	ρ	μ	r	b	Example
20	1	(5, 10)	(2, 1)	(10, 100)	(1, 5)	example 5
20	0.95	(5, 10)	(2, 1)	(10, 100)	(1, 5)	example 6

Table 4.2: Parameters set 2 for Optimal policy

In the first four examples, one class customers are always accepted as long as there are sufficient resources available, while the other type is accepted up to a certain threshold. This is intuitively clear, since we want to reserve resources for the most beneficial customers. However, the example 5 and example 6 show that not all cases have this property.

The advantage of the optimal policy is that it will return the highest reward to the system. We can see from Table 4.1 that the optimal policy performs far better than complete sharing policy in some cases. However, the optimal policy has a few drawbacks. First of all, in cases with a large state space, MDP will have a huge computation complexity which makes it hard to implement in practice. Secondly, based on the results of example 5 (Figure 4.7) and example 6 (Figure 4.8), we can see that the optimal policy may not have an amenable structure. We are therefore motivated to develop other policies that are close to optimal, but at the same time easy to implement.

Due to the fact that the converted system and the original one are not stochastically equivalent, the method for finding reservation policy can be really bad. We now illustrate this with the following examples. The parameters for the original system are given in Table 4.3, the scaled parameters are given in Table 4.4. The optimal policy and the policy obtained through the scaled approximation of example 7 are shown in Figure 4.9 and Figure 4.10 respectively.

B	β	ρ	μ	r	b	R_{Opt}	R_{CS}	R_{Thd}	Example
20	1	(8, 8/7)	(1, 7)	(1, 7)	(1, 7)	47.80	42.35	47.80	example 7
20	1	(8, 8/10)	(1, 10)	(1, 20)	(1, 10)	67.92	8.37	67.92	example 8

Table 4.3: Original system

B	β	ρ	μ	r	b	Rsv	R_{Rsv}
20	1	(8, 8)	(1, 1)	(1, 7)	(1, 1)	(17, 20)	40.96
20	1	(8, 8)	(1, 1)	(1, 20)	(1, 1)	(16, 20)	8.02

Table 4.4: Converted one dimensional system

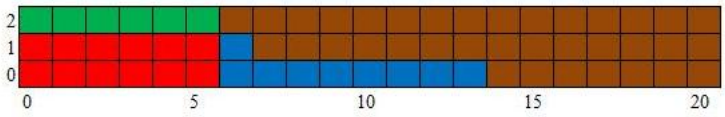


Figure 4.9: Optimal policy of example 7

4.6 Conclusion

In this chapter, we have studied the call admission control policies for the stochastic knapsack problem. We have presented four different types of policies. We have showed the value iteration algorithm to derive the optimal policy. For a complete sharing policy, we derived an explicit expression for calculating the blocking probability of each customer class. Two examples are shown in this chapter to illustrate that the optimal policy does not necessary have an amenable structure. Also, the optimal policy leads to curve of dimensionality. These make the optimal policy practically impossible to implement. Then, we showed other two types of policies: threshold and reservation policies. They were proposed in [11]. The reservation policy does not have a product-form stationary distribution. This makes it difficult to evaluate the performance of the system under any given reservation policy.

A scaling technique was shown to derive reservation policies. However, numerical results show that the returned reservation policies have a bad performance when using such a scaling method. We even found a case where reservation policy is even worse than the complete sharing policy. In general, the threshold policy performs better than the reservation policy. Especially in example 7 and example 8, the reward of the threshold policy is equal to the reward of the optimal policy. However, the reservation policies are more robust than the threshold policies [11].

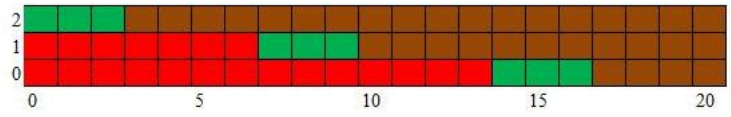


Figure 4.10: Returned Reservation policy of example 7

4.7 Future work

We are interested in finding a better reservation policy as future work. To do so, we need to find an efficient way of calculating the stationary distribution.

Here we present a method to calculate the stationary distribution of a given reservation policy for a problem with a medium size state space (around 5,000 states) without performance evaluation. The method is to calculate the eigenvector corresponding to the eigenvalue 1.

A system under any feasible reservation policy is stable, and therefore has a stationary distribution:

$$\lim_n P_a^n \cdot x_0 = \lim_n x_n = \bar{x}$$

where P_a^n is the transition matrix under reservation policy a, x_0 is any initial state vector, and $\bar{x}$ is the stationary distribution vector. From queueing theory we know that 1 is a eigenvalue of transition matrix P_a with its corresponding eigenvector $\bar{x}$. This result leads us to the thought: *For a transition matrix P, to derive the stationary distribution, we only need to calculate the eigenvector of P_a.*

Regarding to finding eigenvectors, there are multiple ways, i.e. Gauss-Seidel and Newton iteration.

In the stochastic knapsack problem, the transition matrix P_a has the following properties:

- *P_a is a large but sparse matrix.*
- *1 is the dominant eigenvalue of P_a always for aperiodic Markov processes.*

Here we use *Power* method ([3]) to calculate the stationary distribution vector $\bar{x}$. This method is described below:

1. Choose choose a random vector x_0 as the initial vector;
2. $x_{k+1} = \frac{P_a x_k}{\|P_a x_k\|}$;
3. Go to step 2, until $\|x_{k+1} - x_k\| < \epsilon$.

This method will converge linearly if the starting vector x_0 has a nonzero component in the direction of an eigenvector associated with the dominant eigenvalue ([3]). Therefore, as long as the initial vector is correctly chosen, the *Power* method can be implemented in Matlab or *C++* and will converge to the stationary distribution vector.

In cases with a medium state space, using MDP to get the optimal policy already confronts us with a computation problem. The calculation of the eigenvector returns the exact result of the stationary distribution of such medium sized cases. Having the exact result, we can apply **ICSA_RSV** algorithm to find the best reservation policy. However, applying this method to a problem with a large

state space is difficult. This is due to the fact that calculating the eigenvector of a huge matrix is problematic.

Now, the only question is how to derive a reservation policy for a problem with a large state space. Here we will propose a possible method that might be interesting to study. This method is an approximation method to stationary distribution which is easy to implement and compute.

The method is based on the following heuristic. We are interested in finding an easy way to compute the blocking probability. We think that an approximation will yield a nearly optimal reservation policy. Assume there are two classes customers with $\alpha_1 > \alpha_2$. We can calculate $B_1(B)$ and $B_2(B)$, where B stands for the reservation parameter of class two. We also can calculate $B_1(0)$ and $B_2(0)$, where 0 means complete blocking class two customers. Using data $B_1(B)$ and $B_1(0)$, one might use quadratic or exponential functions to estimate $B_1(i)$, which stand for the blocking probability of class 1 customers when set the reservation parameter at i. We do the same for class 2 customers. We think if the close estimation functions were found, it would be easy to compute the a nearly optimal reservation policy.

References

[1] E. Altman, T. Jiménez, and G. Koole. On optimal call admission control in resource-sharing system. *Communications, IEEE Transactions on*, 49(9):1659–1668, 2002.

[2] F. Baccelli, A. Makowski, and A. Shwartz. The fork-join queue and related systems with synchronization constraints: Stochastic ordering and computable bounds. *Advances in Applied Probability*, 21(3):629–660, 1989.

[3] B. Datta. *Numerical linear algebra and applications*. Society for Industrial Mathematics, 2010.

[4] E. Feinberg and M. Reiman. Optimality of randomized trunk reservation. *Probability in the Engineering and Informational Sciences*, 8(04):463–489, 1994.

[5] P. Heidelberger and K. Trivedi. Queueing network models for parallel processing with asynchronous tasks. *Computers, IEEE Transactions on*, 100(11):1099–1109, 2006.

[6] G. Hoekstra, R. van der Mei, and S. Bhulai. Optimal job splitting in parallel processor sharing queues.

[7] J. Kaufman. Blocking in a shared resource environment. *Communications, IEEE Transactions on*, 29(10):1474–1481, 2002.

[8] A. Kumar and R. Shorey. Performance analysis and scheduling of stochastic fork-join jobs in a multicomputer system. *Parallel and Distributed Systems, IEEE Transactions on*, 4(10):1147–1164, 2002.

[9] A. Lebrecht and W. Knottenbelt. Response time approximations in fork-join queues. In *23rd UK Performance Engineering Workshop (UKPEW)*. Citeseer, 2007.

[10] R. Nelson and A. Tantawi. Approximate analysis of fork/join synchronization in parallel queues. *Computers, IEEE Transactions on*, 37(6):739–743, 2002.

[11] J. Ni, D. Tsang, S. Tatikonda, and B. Bensaou. Threshold and reservation based call admission control policies for multiservice resource-sharing systems. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 2, pages 773–783. IEEE, 2005.

[12] J. Ni, D. Tsang, S. Tatikonda, and B. Bensaou. Optimal and structured call admission control policies for resource-sharing systems. *Communications, IEEE Transactions on*, 55(1):158–170, 2007.

[13] R. Ramjee, J. Kurose, D. Towsley, and H. Schulzrinne. Adaptive playout mechanisms for packetized audio applications in wide-area networks. In *INFOCOM'94. Networking for Global Communications., 13th Proceedings IEEE*, pages 680–688. IEEE, 2002.

[14] K. Ross and D. Tsang. The stochastic knapsack problem. *Communications, IEEE Transactions on*, 37(7):740–747, 2002.

[15] K. Ross and D. Yao. Monotonicity properties for the stochastic knapsack. *Information Theory, IEEE Transactions on*, 36(5):1173–1179, 2002.

[16] D. Tsang and K. Ross. Algorithms to determine exact blocking probabilities for multirate tree networks. *Communications, IEEE Transactions on*, 38(8):1266–1271, 2002.

Appendix A: Programming code

Input of the code: N, b, c, M, $\lambda_1, \lambda_2, \cdots, \lambda_N$, $\mu_1, \mu_2, \cdots, \mu_M$.

Output of the code: Input, Aver_sj, Aver_sj_up, Aver_sj_down, Aver_sj_jt[1], Aver_sj_jt_up[1], Aver_sj_jt_down[1], $\cdots$ Aver_sj_jt[N], Aver_sj_jt_up[N], Aver_sj_jt_down[N], Aver_syn, Aver_syn_up, Aver_syn_down, Aver_syn_jt[1], Aver_syn_jt_up[1], Aver_syn_jt_down[1], $\cdots$, Aver_syn_jt[N], Aver_syn_jt_up[N], Aver_syn_jt_down[N].

Input variable	*Description*
N	Number of customer classes
b	Number of sub-job of each class customer
c	Number of overlapping between classes
M	Number of servers
$Aver\_sj$	Average sojourn time of all classes
$Aver\_sj/syn\_up/down$	95% confidence interval
$Aver\_sj\_jt[i]$	Average sojourn time of class i customer
$Aver\_syn$	Average synchronization time of all classes
$Aver\_syn\_jt[i]$	Average synchronization time of class i customer

Table 4.5: Input and output notations

```
#include <iostream>
#include <stdlib.h>
#include <math.h>

using namespace std;

enum kind     {arrival,  subjobdone };

struct event_node {
  double       time;
  kind         type;
  int          id;
  int          qid;
  int          jobtype;
  event_node *next;
}; // struct event_node

class EventList {
  event_node *first;

public:
  EventList();
  ~EventList();
  void schedule_event(double, kind, int, int, int);
  bool get_next_event(double &, kind &, int &, int &, int &);
}; // class EventList

EventList::EventList() {
  first = NULL;
```

```
}; // EventList::EventList


EventList::~EventList() {
  event_node *p;
  while (first != NULL) {
    p = first;
    first = first->next;
    delete p;
  }
}; // EventList::~EventList


void EventList::schedule_event
       (double t, kind soort, int x, int y, int z) {
  event_node *nieuwe_node = new event_node;

  if (nieuwe_node == NULL) {
    cout << "out of memory\n";
    exit(1);
  }; // if

  nieuwe_node->time = t;
  nieuwe_node->type = soort;
  nieuwe_node->id    = x;
  nieuwe_node->qid   = y;
  nieuwe_node->jobtype = z;

  if (first == NULL) {                // lege lijst
    first = nieuwe_node;
    nieuwe_node->next = NULL;
  } // if
  else if (t <= first->time) {        // toevoegen aan begin
    nieuwe_node->next = first;
    first = nieuwe_node;
  }  // else if
  else {
    event_node *p = first;
    event_node *q = NULL;
    while (t > p->time) {             // op juiste plaats tussenvoegen
      q = p;
      p = p->next;
      if (p == NULL)                  // toevoegen aan einde
        break;
    } // while

    q->next = nieuwe_node;
    nieuwe_node->next = p;
  } // else
}; // EventList::schedule_event

bool EventList::get_next_event
```

```
    (double &t, kind &soort, int &x, int &y, int &z) {
  if (first == NULL)                    // lijst leeg
    return false;
  else {
    t = first->time;
    soort = first->type;
    x = first->id;
    y = first->qid;
    z = first->jobtype;

    event_node *p = first;
    first = first->next;
    delete p;
    return true;
  } // else
}; // EventList::get_next_event

#include <iostream>
#include <stdlib.h>
#include <math.h>
#include <cmath>
#include <limits>
#include <cstdlib>
#include <ctime>
#include <fstream>
#include "eventlist2.h"

using namespace std;

const int N = 2000000;
const double endtime = 5000000;
const int batch = 25;
const double T = 1.96;
const int warmup = N/50;              //warm up time

int Njobtype, Noverlap, Nsubtask, Nqueue;
double * lambda, * mu;
int  ** taskmatrix;

EventList evt_list;
double t;
kind soort;
int x;
int y;
int z;
int q;
double RTa [N] = {0};
int Rjobtype [N] = {0};
int RN [N] = {0};
int Rid [N] = {1};
double RTd [N] = {0};
double RTs [40][N] = {0};
```

```
double RTff [N] = {0};
int jobinq [40] = {0};
double simtime = 0;
double total_lambda =100;

// for calculating usage
double final_time = 0;
double batch_time = 0;
int * batch_person_jt [batch];
int batch_start [batch]= {0};
int batch_person [batch] = {0};              // for all computation usage

double total_mean_sj = 0;
double std_sj = 0;
double batch_mean_sj [batch] = {0};       // Average sojourn time
double * batch_mean_sj_jt [batch];
double * total_mean_sj_jt;
double * std_sj_jt;                       // sojourn time for jobtype
double batch_mean_sy [batch] = {0};
double total_mean_sy = 0;
double std_sy = 0;                        // Average synchoronization time

double * batch_mean_sy_jt [batch];
double * total_mean_sy_jt;
double * std_sy_jt;

double draw_exp(double l) {
  long int dummy;

  dummy = rand()+1;
  return -(1/l)*log((double)dummy/(RAND_MAX));
};

int distribution (double total_lambda) {
  double i, k;
  int result, j;
  i = ((double) rand() / (RAND_MAX));
  k = 0;

  for (j=0; j<3; j++) {
      if (i <= ((k+lambda[j]) / total_lambda) && (i > (k/total_lambda))) {
        result = j;
      break;
      }
      k = k + lambda[j];
  }
    return result;
};


int readmatrix(int jt) {
  int j;
```

```
  int NSB = 0;
if (jt > 4) cout<<"Z1:"<<jt<<'\t';

  for (j=0; j<Nqueue; j++) {
    if (taskmatrix [jt][j] > 0) {
          //for the class (type-1), (jth) q
      jobinq [j]++;
          // if taskmatrix[][]=1, a sub job goes to this q
      NSB++;
          // total sub jobs need to be done
        }
  }
  return NSB;
};

int defmatrix () {
  int i, j, ls;
  taskmatrix = new int *[Njobtype];
  for (i=0; i<Njobtype; i++) {
    taskmatrix [i] = new int [Nqueue];
  }                          //decide the size of the taskmatrix
  for (i=0; i<Njobtype; i++) {
    for (j=0; j<Nqueue; j++) {
      taskmatrix [i][j] = 0;
    }
  }                         // initial 0 value for taskmatrix

  ls = 0;
  for (i=0; i<Njobtype; i++) {
    for (j=ls; j<ls+Nsubtask; j++) {
      taskmatrix [i][j] = 1;
    }
    ls = ls - Noverlap + Nsubtask;
  }                        // define the taskmatrix
}


int defpointer () {
  int i, j;

  for (i=0; i<batch; i++) {
    batch_mean_sj_jt[i] =new double [Njobtype];
    batch_mean_sy_jt[i] =new double [Njobtype];
  }
  total_mean_sj_jt = new double [Njobtype];
  std_sj_jt = new double [Njobtype];
  total_mean_sy_jt = new double [Njobtype];
  std_sy_jt = new double [Njobtype];

  for (i=0; i<batch; i++) {
    batch_person_jt[i] = new int [Njobtype];
  }
```

```
  //initial value
  for (i=0; i<Njobtype; i++) {
    total_mean_sj_jt[i] = 0;
    std_sj_jt [i] = 0;
    total_mean_sy_jt[i] = 0;
    std_sy_jt [i] = 0;
    for (j=0; j<batch; j++) {
      batch_mean_sj_jt [j][i] = 0;
      batch_mean_sy_jt [j][i] = 0;
      batch_person_jt [j][i] = 0;
    }
  }
}


void initial () {
  int a, b, i, j;
  total_lambda = 0;
  for (i=0; i<Njobtype; i++) {
    total_lambda = total_lambda + lambda [i];
  }
  a = distribution (total_lambda);
  evt_list.schedule_event(0, arrival, 0, 50, a);
          //(time, eventtype, id, qid, jobtype)
};


void handle_arrival (double t1, kind soort1, int x1, int y1, int z1) {
  int i, j;
  double a = draw_exp (total_lambda);
  int c = distribution (total_lambda);
   RTa [x1] = t1;
   Rid[x1] = x1;
   Rjobtype [x1] = z1;
if (z1 > 4) cout<<"Z1:"<<z1<<'\t';
   RN[x1] = readmatrix (z1);

   for (i=0; i<Nqueue; i++) {
     if (taskmatrix [z1] [i] == 1) {
       if (jobinq [i] == 1) {
       RTs [i][x1] = t1;
       }                          // if there is no one in q0, the
     }                            // current job's arrival time is the
   }                              // start service time;

   for (i=0; i<Nqueue; i++) {
     if (taskmatrix [z1][i] == 1) {
       if (jobinq [i] == 1) {
         double b = draw_exp (mu [i]);
         evt_list.schedule_event (simtime+b, subjobdone, x1, i, z1);
       }
```

```
      }
    }

    if (simtime+a < endtime) {          // schedule the next arrival
      evt_list.schedule_event (simtime+a, arrival, x1+1, 50, c);
    }
};


void handle_subjobdone (double t2, kind soort2, int x2, int y2, int z2) {
  int i, k, j, m;
  double b = draw_exp (mu [y2]);

  if (jobinq [y2] > 1) {
    for (j=x2+1; j<=N; j++) {
      m = Rjobtype [j];
      if (taskmatrix [m][y2] == 1) {
        RTs [y2][j] = t2;
        break;
      }
    }
  }              // record start time for next job in this queue

  if (jobinq[y2] > 1) {
    for (i=x2+1; i<=N; i++) {
      k = Rjobtype [i];
      if (taskmatrix [k][y2] == 1) {
        evt_list.schedule_event (simtime+b, subjobdone, i, y2, k);
        break;
      }
    }
  }      // schedule next subjobdone, if there are more jobs in this q

  if (RTff [x2] ==0) {
    RTff [x2] = t2;
  }
               // record it's first subjob finishing time
  jobinq [y2]--;
  RN [x2]--;

  if (RN [x2] == 0) {
    RTd [x2] = t2;
  }
};


void handle_event (double t, kind soort, int x, int y, int z) {
                     // t=time, soort=eventtype, x=id, y=qid, z=jobtype
  double a;
  int b, c, d;
  simtime = t;
  a = t;
```

```
  b = x;
  c = z;
  d = y;
  switch (soort) {
  case arrival: handle_arrival (a, soort, b, d, c); break;
  case subjobdone: handle_subjobdone (a, soort, b, d, c); break;
  }
};


void cal_Aver_sj () {
  int i, j;

  if (RTd [N-1] > 0) {
    final_time = RTd [N-1] - RTd[warmup];
    batch_time = final_time / batch;
  }

  for (j=0; j<batch; j++) {
    for (i=warmup; i<N; i++) {
      if (RTd [i] >= (batch_time + batch_time * j)) {
        batch_person [j] = i -1;
        break;
      }
    }
  }
  batch_person [batch-1] = N-1;

  for (j=0; j<batch; j++) {
      if (j == 0) {
        batch_start [j] = warmup;
      }
      else {batch_start [j] = batch_person [j-1] + 1;}

      for (i=batch_start [j]; i<(batch_person [j]+1); i++) {
        batch_mean_sj [j] = batch_mean_sj [j] + (RTd [i] - RTa [i]);
      }
      batch_mean_sj [j] = batch_mean_sj [j] /
                          (double)(batch_person[j] - batch_start[j] + 1);
  }

  total_mean_sj = 0;
  for (i=0; i<batch; i++) {
    total_mean_sj = total_mean_sj + batch_mean_sj [i];
  }
  total_mean_sj = total_mean_sj / (double)batch;

  for (i=0; i<batch; i++) {
    std_sj =std_sj + (batch_mean_sj[i] - total_mean_sj) *
            (batch_mean_sj[i] - total_mean_sj);
  }            // calculate the Average sojourn time (with CI 95%)
}
```

```
void cal_sj_jt () {
  int i, j, k;

  for (k=0; k<Njobtype; k++) {
    for (i=0; i<batch; i++) {
      for (j=batch_start[i]; j<=batch_person[i]; j++) {
        if (Rjobtype[j] == k) {
          batch_mean_sj_jt [i][k] =
          batch_mean_sj_jt[i][k] + (RTd[j] - RTa[j]);
          batch_person_jt [i][k]++;
        }
      }
      batch_mean_sj_jt[i][k] = batch_mean_sj_jt[i][k] /
                               (double)(batch_person_jt[i][k]);
    }
  }                          // calculate batch mean for each jobtype

  for (j=0; j<Njobtype; j++) {
    for (i=0; i<batch; i++) {
      total_mean_sj_jt[j] = total_mean_sj_jt[j] + batch_mean_sj_jt[i][j];
    }
    total_mean_sj_jt[j] = total_mean_sj_jt[j] / (double)(batch);
  }                        // calculate total mean of batches for each jobtype

  for (j=0; j<Njobtype; j++) {
    for (i=0; i<batch; i++) {
      std_sj_jt[j] = std_sj_jt[j] + (batch_mean_sj_jt [i][j] -
         total_mean_sj_jt[j])*
          (batch_mean_sj_jt [i][j] - total_mean_sj_jt[j]);
    }
  }      // calculate w (which similar to std) for each jobtype
}

void cal_Aver_sy () {
  int i, j;

  for (i=0; i<batch; i++) {
    for (j=batch_start[i]; j<=batch_person[i]; j++) {
      if (RTa [j] > RTd [j]) { cout<<"haha, fault"<<endl;}
      batch_mean_sy[i] = batch_mean_sy[i] + (RTd [j] - RTff [j]);
    }
    batch_mean_sy[i] = batch_mean_sy[i] /
              (double)(batch_person[i] - batch_start[i] + 1);
                       //calculate total average sychronization time
  }

  for (i=0; i<batch; i++) {
    total_mean_sy = total_mean_sy + batch_mean_sy [i];
  }
  total_mean_sy = total_mean_sy / (double)(batch);
```

```
  for (i=0; i<batch; i++) {
    std_sy = std_sy + (batch_mean_sy[i] - total_mean_sy) *
             (batch_mean_sy[i] - total_mean_sy);
  }
}

void cal_sy_jt () {
  int i, j, k;
  for (i=0; i<batch; i++) {
    for (j=0; j<Njobtype; j++) {
      for (k=batch_start[i]; k<=batch_person[i]; k++) {
        if (Rjobtype[k] == j) {
          batch_mean_sy_jt[i][j] = batch_mean_sy_jt[i][j]
                                   + (RTd [k] - RTff [k]);
        }
      }
      batch_mean_sy_jt[i][j] = batch_mean_sy_jt[i][j] /
                               (double)(batch_person_jt[i][j]);
                  // average Synchronization time for each jobtype
    }
  }

  for (i=0; i<batch; i++) {
    for (j=0; j<Njobtype; j++) {
      total_mean_sy_jt[j] = total_mean_sy_jt[j]
                          + batch_mean_sy_jt[i][j];
    }
  }
  for (i=0; i<Njobtype; i++) {
    total_mean_sy_jt[i] = total_mean_sy_jt[i] / (double)batch;
  }

  for (i=0; i<batch; i++) {
    for (j=0; j<Njobtype; j++) {
   std_sy_jt[j] = std_sy_jt[j] + (batch_mean_sy_jt[i][j]-total_mean_sy_jt[j])
                      *(batch_mean_sy_jt[i][j]-total_mean_sy_jt[j]);
    }
  }
}

void report() {
  int i, j;
  cout <<"Rid"<<'\t'<<"RTa"<<'\t'<<"RTd"<<endl;
  for (i=0; i<N; i++) {
    cout <<Rid[i]<<'\t'<<RTa [i]<<'\t'<<RTd [i]<<endl;
  }
};

void del() {
  int i;
```

```
  for (i=0; i<Njobtype; i++) {
    delete [] taskmatrix[i];
  }
  delete [] taskmatrix;
  delete [] lambda;
  delete [] mu;
  for (i=0; i<batch; i++) {
    delete [] batch_mean_sj_jt[i];
  }
  delete [] total_mean_sj_jt;
  delete [] std_sj_jt;
  for (i=0; i<batch; i++) {
    delete [] batch_person_jt[i];
  }
}

void output () {
  int i, j;

  cout << Njobtype<<";" << Nsubtask<<";"<<Noverlap<<";"<<Nqueue<<";";
  for (i=0; i<Njobtype; i++) {
    cout<<lambda[i]<<";";
  }
  for (j=0; j<Nqueue; j++) {
    cout<<mu[j]<<";";
  }                              // output lambda and mu

   cout<<total_mean_sj<<";"<<total_mean_sj+T*
    sqrt (std_sj / (double)(batch*(batch-1)))<<";"
   <<total_mean_sj-T*sqrt (std_sj / (double)(batch*(batch-1)))<<";";
                                 // output total average sojourn time

  for (i=0; i<Njobtype; i++) {
    cout<<total_mean_sj_jt[i]<<";"<<total_mean_sj_jt[i] +
          T*sqrt(std_sj_jt[i] / (double)(batch*(batch-1)))<<";"
        <<total_mean_sj_jt[i] -
          T*sqrt(std_sj_jt[i] / (double)(batch*(batch-1)))<<";";
  }                               // output sojourn time for each jobtype

  cout<<total_mean_sy<<";"<<total_mean_sy+T*sqrt(std_sy /
                               (double)(batch*(batch-1)))<<";"
      <<total_mean_sy-T*sqrt(std_sy / (double)(batch*(batch-1)))<<";";

 for (i=0; i<Njobtype; i++) {
    cout<<total_mean_sy_jt[i]<<";"<<total_mean_sy_jt[i] +
          T*sqrt(std_sy_jt[i] / (double)(batch*(batch-1)))<<";"
        <<total_mean_sy_jt[i] - T*sqrt(std_sy_jt[i] /
                                 (double)(batch*(batch-1)))<<";";
 }
  cout<<endl;
}
```

```
int main(int argc, char* argv[]) {
  int i, j = 0, d;
  int s;

  if (argc >= 5) {
    d = atoi(argv[1]);
    s  = atoi(argv[4]);
    lambda = new double [d];
    mu = new double [s];

    if (argc >= 5 + d + s) {
      Njobtype =atoi(argv[1]);
      Nsubtask = atoi(argv[2]);
      Noverlap = atoi(argv[3]);
      Nqueue =  atoi(argv[4]);

      for (i=0; i<d; i++) {
        lambda [i] = atof(argv[5+i]);
      }
      for (j=0; j<s; j++) {
        mu [j] = atof(argv [5+d+j]);
      }
    }             // give value to mu and lambda;
    else {cout<<"More input, man!!!!"<<endl;}
  }

  else {
    Njobtype = 3;
    Nsubtask = 1;
    Noverlap = 0;
    Nqueue = 3;
    lambda = new double [Njobtype];
    mu = new double [Nqueue];
    for (i=0; i<Njobtype; i++) {
      lambda [i] = 1;
    }
    for (i=0; i<Nqueue; i++) {
      mu [i] = 2;
    }
  }             // in case the input is not enough

  defmatrix ();
  defpointer ();
  initial ();
  srand((unsigned)time(NULL));

  while (evt_list.get_next_event (t, soort, x, y, z)) {
    handle_event (t, soort, x, y, z);
  }

  cal_Aver_sj ();
  cal_sj_jt ();
```

```
    cal_Aver_sy ();
    cal_sy_jt ();

    output ();
    del();
    return 0;
}
```

Appendix B: Simulation and verification

We will present the idea of how to simulate our model. The programming used is *C++*. We use event based simulation, i.e. events (arrivals or departures) will occur after exponentially distributed time. We simulate 2 million jobs in total. For each job, 5 types of data are recorded in the simulation: $T_a, T_{ff}, T_d, T_{soj}$ and T_{syn} (Table 4.6).

Variable	*Description*
T_a	Arrival time of jobs
T_{ff}	Time that the first sub-job finishes
T_d	Time that a job leaves the system
T_{soj}	The sojourn time of a job
T_{syn}	The time a job spends in the synchronization buffer

Table 4.6: Variables

According to the definition, we have:

$$T_{soj} = T_d - T_a,$$

$$T_{syn} = T_d - T_{ff}$$

For these 2 million jobs, we divide the data into 26 segments. The first segment includes $\frac{2\ million}{50}$ jobs. We will not use these data, because the system is not stable at the beginning of the simulation. Each of the other segments consists of $\frac{2\ million - 2\ million/50}{25}$ jobs. We calculate the mean of each segment, denoted as $me_i, i = 2, 3, \ldots, 26$. The total mean Me can be calculated by:

$$Me = \frac{\sum_{i=2}^{26} me_i}{25},$$

with a 95% confidence interval $(Me + 1.96 \times \sqrt{\frac{W}{25\times(25-1)}},\ Me - 1.96 \times \sqrt{\frac{W}{25\times(25-1)}})$

where $W = \sum_{i=2}^{26}(me_i - Me)$.

Here we show some verification of multi-class Fork-Join model through few examples. For each example, we compare the simulation result and exact solution.

example	exact	sim
M/M/1 ($\lambda = 1, \mu = 2$)	1	1.00408
M/M/1 ($\lambda = 1, \mu = 1.1$)	10	10.4437
M/M/1 ($\lambda = 10, \mu = 20$)	0.1	0.100024

Table 4.7: Sojourn time verification

Appendix C: Stochastic knapsack problem MDP

```
#include <iostream>
#include <stdlib.h>
#include <math.h>
#include <cmath>
#include <limits>
#include <cstdlib>
#include <ctime>
#include <fstream>

#define min(a,b) (((a) < (b)) ? (a) : (b))
#define max(a,b) (((a) > (b)) ? (a) : (b))

using namespace std;

const int Njobtype = 2;
const int L = 100000000;
const double beta = 1;
const double epsilon = 1.0E-8;

int B;
double tao;
int Niterate = 0;
double lambda[Njobtype];
double mu[Njobtype];
int reward[Njobtype];
int request[Njobtype];
int row, col;

double ** Vn;
double ** Vnp1;
int ** opt_policy;

void initialize(void){
  int i, j;
  row = (int) (B/request[0]) + 1;
  col = (int) (B/request[1]) + 1;

  Vn = new double * [row];
  Vnp1 = new double * [row];
  opt_policy = new int * [row];

  for (i=0; i<row; i++) {
    Vn [i] = new double [col];
    Vnp1 [i] = new double [col];
    opt_policy [i] = new int [col];
 }

  for (int x1 = 0; x1 < row; x1++) {
    for (int x2 = 0; x2 < col; x2++) {
```

```
      Vn [x1][x2] = 0.0;
      Vnp1 [x1][x2] = 0.0;
      opt_policy [x1][x2] = 0.0;
    }
  }
}


void iterate(void) {
  for (int x1 = 0; x1 < row; x1++) {
    for (int x2 = 0; x2 < col; x2++) {
      Vn [x1][x2] = Vnp1 [x1][x2];
    }
  }
}


double V(int x1, int x2) {
  int i, j;
  double C, value;
  double arrival_term1, arrival_term2, arrival_term;
  double departure_term1 = 0.0, departure_term2 = 0.0;
  double departure_term = 0.0;
  double uniform_term = 0;
  double rate = 0.0;

  if (  (x1*request[0]+x2*request[1]) > B ) {
    C = L;
  }
  else {
    C = 0;
  }

  if (x1 == row-1) {
    arrival_term1 = lambda[0] *Vn[x1][x2];
  }
  else {
    arrival_term1 = lambda[0] * max (Vn[x1][x2], Vn[x1+1][x2]+reward[0]);
  }

  if (x2 == col-1) {
    arrival_term2 = lambda[1] * Vn[x1][x2];
  }
  else {
    arrival_term2 = lambda[1] *max (Vn[x1][x2], Vn[x1][x2+1]+reward[1]);
  }
  arrival_term = beta * (arrival_term1 + arrival_term2);
  rate = (lambda[0] + lambda[1]);

  if (x1 == 0) {
    departure_term1 =(double) (int)(B/request[0]) *Vn[x1][x2];
    rate = rate + mu[0] * (int)(B/request[0]);
```

```
  }
  else {
    departure_term1 =(double) x1 *(Vn [x1-1][x2])
                    +(double) (((int)(B/request[0]) - x1)) *(Vn[x1][x2]);
    rate = rate + mu[0] * (int)(B/request[0]);
  }
  departure_term1 = departure_term1 * mu[0];

  if (x2 == 0) {
    departure_term2 =(double)((int)(B/request[1])) * Vn[x1][x2];
    rate = rate + mu[1] * (int)(B/request[1]);
  }
  else {
    departure_term2 = (double) x2 * Vn [x1][x2-1] +
                      (double) (((int)(B/request[1]) - x2))
                      * (Vn[x1][x2]);
    rate = rate + mu[1] * (int)(B/request[1]);
  }
  departure_term2 = departure_term2 * (mu[1]);
  departure_term = beta * (departure_term1 + departure_term2);

  uniform_term = beta * (tao - rate) * Vn[x1][x2];
  value = -C + arrival_term + departure_term + uniform_term
;
  value = value / tao;

  return value;
}


double optpolicy(int x1, int x2) {

  double temp0, temp1, temp2, temp3;
  int policy;
  double a, b, c;

  if (x1 * request[0] + x2 * request[1] >= B) {
    policy = 0;
  }
  else if ((x1+1)*request[0]>B && (x2+1)*request[1]>B) {
    policy = 0;
  }
  else if ((x1+1)*request[0]<=B && (x2+1)* request[1]>B) {
    temp0 = lambda[0] * Vnp1[x1][x2] + lambda[1] * Vnp1[x1][x2];
    temp1 = lambda[0] * (Vnp1[x1+1][x2] + reward[0]) +
            lambda[1] * Vnp1[x1][x2];
    if (temp0 > temp1) {
      policy = 0;
    }
    else {
      policy = 1;
    }
```

```
  }
  else if ((x1+1)*request[0]>B && (x2+1)*request[1]<=B) {
    temp0 = lambda[0] * Vnp1[x1][x2] +
            lambda[1] * Vnp1[x1][x2];
    temp2 = lambda[0] * Vnp1[x1][x2] +lambda[1] *
            (Vnp1[x1][x2+1] + reward[1]);
    if (temp0>temp2) {
      policy = 0;
    }
    else {
      policy = 2;
    }
  }
  else {
    temp0 = lambda[0] * Vnp1[x1][x2] + lambda[1] * Vnp1[x1][x2];
    temp1 = lambda[0] * (Vnp1[x1+1][x2]+reward[0]) +
            lambda[1] * Vnp1[x1][x2];
    temp2 = lambda[0] * Vnp1[x1][x2] + lambda[1] *
            (Vnp1[x1][x2+1] + reward[1]);
    temp3 = lambda[0] * (Vnp1[x1+1][x2]+reward[0]) + lambda[1] *
            (Vnp1[x1][x2+1]+reward[1]);
    a = max(temp0, temp1);
    b = max(temp2, temp3);
    c = max(a, b);
    if (c == temp0) { policy = 0;}
   else if (c == temp1) { policy = 1;}
   else if (c == temp2) { policy = 2;}
   else { policy = 3;}
  }
  return policy;
}

void valueiteration(void) {
  double Mn, mn;

  do {
    Mn = -10000;
    mn =  10000;

    Niterate ++;
    iterate();
    for (int x1 = 0; x1 <= row-1; x1++) {
      for (int x2 = 0; x2 <= col-1; x2++) {
        Vnp1 [x1][x2] = V(x1, x2);
        Mn = max(Vnp1[x1][x2] - Vn[x1][x2], Mn);
        mn = min(Vnp1[x1][x2] - Vn[x1][x2], mn);
      }
    }
  } while (Mn - mn > epsilon);
  cout<< (Mn + mn)*tao / 2.0 <<endl;
}
```

```
void setparameter (int B_total, double LAMBDA_1, double LAMBDA_2,
                   double MU_1, double MU_2,
                   int R_1, int R_2, int b_1, int b_2) {

  B = B_total;
  lambda[0] = LAMBDA_1;
  lambda[1] = LAMBDA_2;
  mu[0] = MU_1;
  mu[1] = MU_2;
  reward[0] = R_1;
  reward[1] = R_2;
  request[0] = b_1;
  request[1] = b_2;

  tao  = lambda[0] + lambda[1] + (double) ((int)(B / request[0])) * mu[0] +
         (double) ((int)(B / request[1])) * mu[1];
  cout<<tao<<endl;

  cout<<B<<";"<<lambda[0]<<";"<<lambda[1]<<";"<<mu[0]<<";"<<mu[1]<<";"
      <<reward[0]<<";"<<reward[1]<<";"<<request[0]<<";"<<request[1]<<';';
}

void del () {
  int i;
  for (i=0; i<row; i++) {
    delete [] Vn [i];
    delete [] Vnp1 [i];
    delete [] opt_policy [i];
  }
  delete [] Vn;
  delete [] Vnp1;
  delete [] opt_policy;
}


int main(int argc, char* argv[]) {
  int x1, x2;

  if (argc < 10) {
    cout<<"more input is required!"<<endl;
  }
  else if (argc > 10) {
    cout<<"too much input, I can't handle!!!"<<endl;
  }
  else setparameter (atoi(argv[1]), atof(argv[2]), atof(argv[3]),
                     atof(argv[4]), atof(argv[5]), atoi(argv[6]),
                     atoi(argv[7]), atoi(argv[8]), atoi(argv[9]));
  initialize();
  valueiteration();

   for (int x1 = 0; x1 <= row-1; x1++) {
      for (int x2 = 0; x2 <= col-1; x2++) {
```

```
        opt_policy [x1][x2] = optpolicy (x1, x2);
      }
    }
  for (int x1 = 0; x1<row; x1++) {

     for (int x2 = 0; x2 < col; x2++) {
       if (x2 == col-1) {
         cout<<opt_policy [x1][x2];
       }
       else {
       cout<<opt_policy [x1][x2] <<"     ";
       }
     }
     cout<<endl;
   }
  del ();

  return 0;
}
```